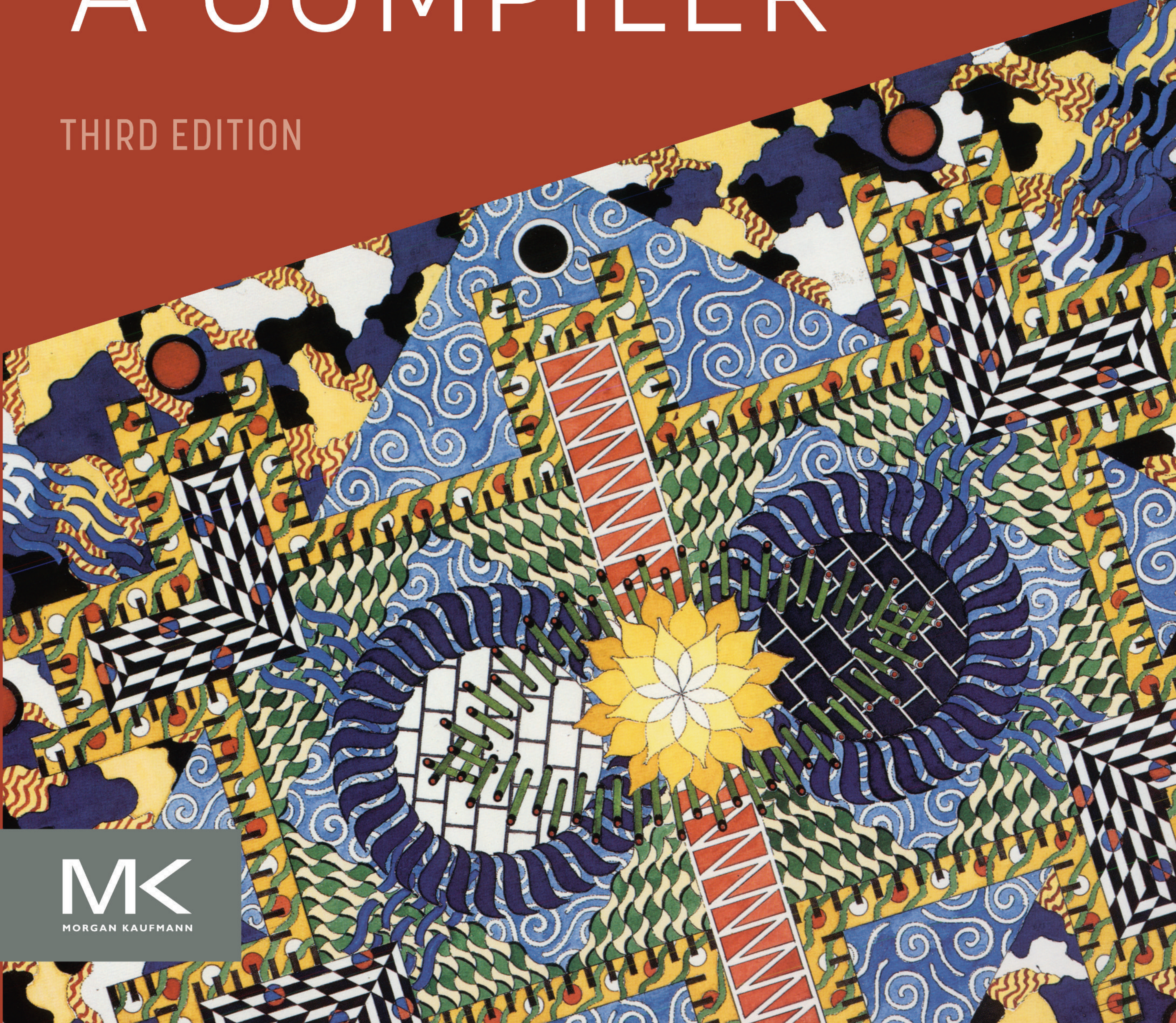
Keith D. Cooper & Linda Torczon

ENGINEERING A COMPILER

THIRD EDITION



MK MORGAN KAUFMANN

Engineering a Compiler

This page intentionally left blank

Engineering a Compiler

Third Edition

Keith D. Cooper

Linda Torczon

ELSEVIER

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER

Cover Image: “The Landing of the Ark,” a vaulted ceiling-design whose iconography was narrated, designed, and drawn by John Outram of John Outram Associates, Architects and City Planners, London, England. To read more, visit www.johnoutram.com/rice.html.

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

ISBN: 978-0-12-815412-0

For information on all Morgan Kaufmann publications
visit our website at https://www.elsevier.com/books-and-journals

Publisher: Katey Birtcher
Acquisitions Editor: Steve Merken
Editorial Project Manager: Rebecca Gruliow; Beth LoGiudice
Production Project Manager: Manchu Mohan
Designer: Margaret Reid

Typeset by VTeX

Printed in The United States of America

Last digit is the print number:
9 8 7 6 5 4 3 2 1

We dedicate this volume to

- *our parents, who instilled in us the thirst for knowledge and supported us as we developed the skills to follow our quest for knowledge;*
- *our children, who have shown us again how wonderful the process of learning and growing can be; and*
- *our spouses, without whom this book would never have been written.*

This page intentionally left blank

Contents

This page intentionally left blank

About the Authors

Keith D. Cooper is the Doerr Professor in Computational Engineering at Rice University. He has worked on a broad collection of problems in the optimization of compiled code, including interprocedural data-flow analysis and its applications, value numbering, algebraic reassociation of expressions, register allocation, and instruction scheduling. He has taught a variety of courses, including introductory programming, algorithmic thinking, computer organization, team programming, several versions of a compiler construction course, and a graduate course on code optimization. He has served as Chair of Rice's Computer Science Department, Chair of its Computational and Applied Mathematics Department, and Associate Dean for Research in Rice's Engineering School. He is a Fellow of the ACM.

Linda Torczon had a long career as a Senior Research Scientist in Rice University's Computer Science Department. She has worked on a broad variety of topics in compilation, including interprocedural data-flow analysis, inline substitution, register allocation, adaptive optimization, and the design of programming environments. She taught Rice's undergraduate compiler course several times. Dr. Torczon served as Executive Director of the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center. She also served as the Executive Director of HiPerSoft, of the Los Alamos Computer Science Institute, and of the Virtual Grid Application Development Software Project. She was a principal investigator on the DARPA-sponsored Platform Aware Compilation Environment project, which designed an optimizing compiler that could automatically adjust its optimizations and strategies to new platforms. She also served as the director of Rice's Computer Science Professional Master's Degree program.

This page intentionally left blank

About the Cover

The cover of this book features a portion of the drawing "The Landing of the Ark," which decorates the ceiling of Duncan Hall at Rice University. Both Duncan Hall and its ceiling were designed by British architect John Outram (principal of John Outram Associates in London, England). Duncan Hall is an outward expression of architectural, decorative, and philosophical themes developed over Outram's career as an architect. The decorated ceiling of the ceremonial hall plays a central role in the building's decorative scheme. Outram inscribed the ceiling with a set of significant ideas—a creation myth. By expressing those ideas in an allegorical drawing of vast size and intense color, Outram created a signpost that tells visitors who wander into the hall that, indeed, this building is not like other buildings.

By using the same signpost on the cover of *Engineering a Compiler*, the authors intend to signal that this work contains significant ideas that are at the core of their discipline. Like Outram's building, this volume is the culmination of intellectual themes developed over the authors' professional careers. Like Outram's decorative scheme, this book is a device for communicating ideas. Like Outram's ceiling, it presents significant ideas in new ways.

By connecting the design and construction of compilers with the design and construction of buildings, we intend to convey the many similarities in these two distinct activities. Our many long discussions with Outram introduced us to the Vitruvian ideals for architecture: commodity, firmness, and delight. These ideals apply to many kinds of construction. Their analogs for compiler construction are consistent themes of this text: function, structure, and elegance. Function matters; a compiler that generates incorrect code is useless. Structure matters; engineering details determine a compiler's efficiency and robustness. Elegance matters; a well-designed compiler, in which the algorithms and data structures flow smoothly from one pass to another, can be a thing of beauty.

We are delighted to have John Outram's work grace the cover of this book.

Duncan Hall's ceiling is an interesting technological artifact. Outram drew the original design on one sheet of paper. It was photographed and scanned at 1200 DPI yielding roughly 750 MB of data. The image was enlarged to form 234 distinct 2 × 8 foot panels, creating a 52 × 72 foot image. The panels were printed onto oversized sheets of perforated vinyl using a 12-DPI acrylic-ink printer. These sheets were precision-mounted onto 2 × 8 foot acoustic tiles and hung on the vault's aluminum frame. For more information, see: www.johnoutram.com/rice.html.

This page intentionally left blank

Preface

CHANGES IN THE THIRD EDITION

The changes introduced in the third edition of *Engineering a Compiler* (EaC3e) arise from two principal sources: changes in the way that programming-language translation technology is used and changes in the technical backgrounds of our students. These two driving forces have led to continual revision in the way that we teach compiler construction. This third edition captures those classroom-tested changes.

EaC3e reorganizes the material, discarding some topics from prior editions and introducing some topics that are new—at least in this book. EaC2e included major changes to the material on optimization (Chapters 8, 9, and 10); those chapters are largely intact from the second edition. For EaC3e, we have made changes throughout the book, with a particular focus on reorganization and revision in the middle of the book: Chapters 4 through 7. The widespread use of just-in-time (JIT) compilers prompted us to add a chapter to introduce these techniques. In terms of specific content changes:

- The chapter on intermediate representations now appears as Chapter 4, before syntax-driven translation. Students should be familiar with that material before they read about syntax-driven translation.
- The material on attribute grammars that appeared in Chapter 4 of the first two editions is gone. Attribute grammars are still an interesting topic, but it is clear that ad-hoc translation is the dominant paradigm for translation in a compiler's front end.
- Chapter 5 now provides a deeper coverage of both the mechanism of syntax-driven translation and its use. Several translation-related topics that were spread between Chapters 4, 6, and 7 have been pulled into this new chapter.
- Chapter 7 has a new organization and structure, based on extensive in-class experimentation.
- Chapter 13 now focuses on two allocators: a local allocator based on Best's algorithm and a global allocator based on the work of Chaitin and Briggs. The Advanced Topics section of Chapter 13 explores modifications of the basic Chaitin-Briggs scheme that produce techniques such as linear scan allocators, SSA-based allocators, and iterative coalescing allocators.

Earlier editions called Best's algorithm the "bottom-up local" algorithm.

- Chapter 14 provides an overview of runtime optimization or JIT-compilation. JIT compilers have become ubiquitous. Students should understand how JIT compilers work and how they relate to classic *ahead-of-time* compilers.

Our goal continues to be a text and a course that expose students to critical issues in modern compilers and provide them with the background to tackle those problems.

ORGANIZATION

EaC3e divides the material into four roughly equal pieces:

- The first section, Chapters 2 and 3, covers the design of a compiler's front end and the algorithms that go into tools that automate front-end construction. In teaching the algorithms to derive scanners from regular expressions and parsers from context-free grammars, the text introduces several key concepts, including the notion of a fixed-point algorithm.
- The second section, Chapters 4 through 7, explores the mapping of source code into the compiler's intermediate form. These chapters examine the kinds of code that the front end can generate for the optimizer and the back end.
- The third section, Chapters 8, 9, 10, and 14, presents an overview of code optimization. Chapter 8 provides a broad look at the kinds of optimization that compilers perform. Chapters 9 and 10 dive more deeply into data-flow analysis and scalar optimization. Chapter 14 fits thematically into this section; it assumes knowledge of the material in the fourth section.
- The fourth section, Chapters 11 through 13, focuses on the major algorithms found in a compiler's back end: instruction selection, instruction scheduling, and register allocation. In the third edition, we have revised the material on register allocation so that it focuses on fewer ideas and covers them at greater depth. The new chapter provides students with a solid basis for understanding most of the modern allocation algorithms.

We usually omit Chapters 9 and 10 in the undergraduate course. Chapter 14 elicits, in our experience, more student interest.

Our undergraduate course takes a largely linear walk through this material. We often omit Chapters 9 and 10 due to a lack of time. The material in Chapter 14 was developed in response to questions from students in the course.

APPROACH

Compiler construction is an exercise in engineering design. The compiler writer must choose a path through a design space that is filled with diverse alternatives, each with distinct costs, advantages, and complexity. Each decision has an impact on the resulting compiler. The quality of the end product depends on informed decisions at each step along the way.

Thus, there is no single right answer for many of the design decisions in a compiler. Even within "well-understood" and "solved" problems, nuances in design and implementation have an impact on both the behavior of the compiler and the quality of the code that it produces. Many considerations play into each decision. As an example, the choice of an intermediate representation for the compiler has a profound impact on the rest of the compiler, from time and space requirements through the ease with which different algorithms can be applied. The decision, however, is often given short shrift. Chapter 4 examines the space of intermediate representations and some of the issues that should be considered in selecting one. We raise the issue again at several points in the book—both directly in the text and indirectly in the exercises.

EaC3e explores the compiler construction design space and conveys both the depth of problems and the breadth of the possible solutions. It presents some of the ways that problems in compilation have been solved, along with the constraints that made those solutions attractive. Compiler writers need to understand both the parameters of the problems and their solutions. They must recognize that a solution to one problem can affect both the opportunities and constraints that appear in other parts of the compiler. Only then can they make informed and intelligent design choices.

PHILOSOPHY

This text exposes our philosophy for building compilers, developed during more than forty years each of research, teaching, and practice. For example, intermediate representations should expose those details that matter in the final code; this belief leads to a bias toward low-level representations. Values should reside in registers until the allocator discovers that it cannot keep them there; this practice leads to compilers that operate in terms of virtual registers and that only store values to memory when it cannot be avoided. This approach also increases the importance of effective algorithms in the compiler's back end. Every compiler should include optimization; it simplifies the rest of the compiler. Our experiences over the years have informed the selection of material and its presentation.

A WORD ABOUT PROGRAMMING EXERCISES

A class in compiler construction offers the opportunity to explore complex problems in the context of a concrete application—one whose basic functions are well understood by any student with the background for a compiler construction course. In most versions of this course, the programming exercises play a large role.

We have taught this class in versions where the students build a simple compiler from start to finish—beginning with a generated scanner and parser and ending with a code generator for some simplified RISC instruction set. We have taught this class in versions where the students write programs that address well-contained individual problems, such as register allocation or instruction scheduling. The choice of programming exercises depends heavily on the role that the course plays in the surrounding curriculum.

In some schools, the compiler course serves as a capstone course for seniors, tying together concepts from many other courses in a large, practical, design and implementation project. Students in such a class might write a complete compiler for a simple language or modify an open-source compiler to add support for a new language feature or a new architectural feature. This version of the class might present the material in a linear order that closely follows the text's organization.

In some schools, that capstone experience occurs in other courses or in other ways. In this situation, the teacher might focus the programming exercises more narrowly on algorithms and their implementations, using labs such as a local register allocator or a tree-height rebalancing pass. This version of the course might skip around in the text and adjust the order of presentation to meet the needs of the labs. We have found that students entering the course understand assembly-language programming, so they have no problem understanding how a scheduler or a register allocator should work.

In either scenario, the course should make connections to other classes in the undergraduate curriculum. The course has obvious ties to computer organization, assembly-language programming, operating systems, computer architecture, algorithms, and formal languages. Less obvious connections abound. The material in Chapter 7 on character copying raises performance issues that are critical in protocol-stack implementation. The material in Chapter 2 has applications that range from URL-filtering through specifying rules for firewalls and routers. And, of course, Best's algorithm from Chapter 13 is an earlier version of Belady's offline page replacement algorithm, MIN.

ADDITIONAL MATERIALS

Additional resources are available to help you adapt the material presented in EaC3e to your course. These include a complete set of slides from the authors' version of the course at Rice University and a set of solutions to the exercises. Visit https://educate.elsevier.com/book/details/9780128154120 for more information.

ACKNOWLEDGMENTS

Many people were involved in the preparation of this third edition of *Engineering a Compiler*. People throughout the community have gently pointed out inconsistencies, typographical problems, and errors. We are grateful to each of them.

Teaching is often its own reward. Two colleagues of ours from the classroom, Zoran Budimlić and Michael Burke, deserve special thanks. Zoran is a genius at thinking about how to abstract a problem. Michael has deep insights into the theory behind both the front-end section of the book and the optimization section. Each of them has influenced the way that we think about some of this material.

The production team at Elsevier, specifically Beth LoGiudice, Steve Merken, and Manchu Mohan, played a critical role in the conversion of a rough manuscript into its final form. All of these people improved this volume in significant ways with their insights and their help. Aaron Keen, Michael Lam, and other reviewers provided us with valuable and timely feedback on Chapter 14.

Finally, many people have provided us with intellectual and emotional support over the last five years. First and foremost, our families and our colleagues at Rice have encouraged us at every step of the way. Christine and Carolyn, in particular, tolerated myriad long discussions on topics in compiler construction. Steve Merken guided this edition from its inception through its publication with enthusiasm, extreme patience, and good humor. To all these people go our heartfelt thanks.

This page intentionally left blank

Chapter 1

Overview of Compilation

ABSTRACT

A compiler is a computer program that translates a program written in one language into a program written in another language. At the same time, a compiler is a large software system with many internal components and algorithms. These components interact in complex ways. The study of compiler construction is both an introduction to techniques for the translation and improvement of programs, and a practical exercise in software engineering. This chapter provides a conceptual overview of all the major components of a modern compiler.

KEYWORDS

Compiler, Interpreter, Automatic Translation

1.1 INTRODUCTION

The role of the computer in daily life grows each year. Computers play critical roles in entertainment, communication, transportation, medical devices, design, manufacturing, and even mundane devices such as household thermostats. Computation has created entirely new categories of activity, from video games to social networks. It has changed the way that we predict weather and that we receive the news. Embedded computers synchronize traffic lights and deliver e-mail to your pocket.

Almost all computer applications rely on software—computer programs that build virtual tools on top of the low-level abstractions provided by the underlying hardware. Almost all of that software is translated by a tool called a *compiler*. A compiler is simply a computer program that translates other computer programs to prepare them for execution. This book presents the fundamental techniques of automatic translation that are used in compiler construction. It describes many of the challenges that compiler writers face and the algorithms that they use to address them.

Compiler
a computer program that translates other computer programs

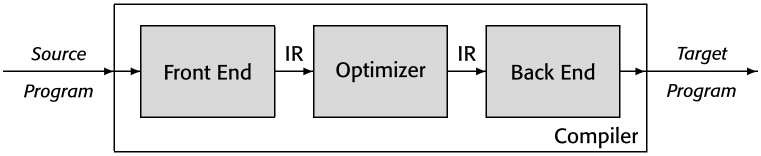
In the text, we use the terms “code” and “program” to mean either some part of a program or the whole program. The meaning should be clear from context.

Conceptual Roadmap

A compiler translates software written in one language into another language. To translate text from one language to another, the tool must un-

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00007-3

derstand both the form, or syntax, and content, or meaning, of the input language. It needs to understand the rules that govern syntax and meaning in the output language. Finally, it needs a scheme for mapping content from the source language to the target language.



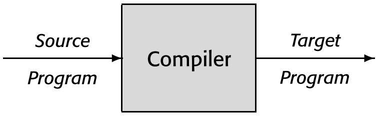
The structure of a typical compiler derives from these simple observations. The compiler has a front end to deal with the source language. It has a back end to deal with the target language. Connecting the front end and the back end, it has a formal structure for representing the program in an intermediate representation (IR), whose meaning is largely independent of either language. Most compilers include an optimizer that sits between the front end and the back end; the optimizer analyzes the IR and rewrites it into a form that is, under one or more metrics, better.

Overview

Computer programs are simply sequences of abstract operations written in a *programming language*—a formal language designed for expressing computation. Programming languages have rigid properties and meanings—in contrast with natural languages, such as Chinese, Portuguese, or English. Programming languages are designed for expressiveness, conciseness, and clarity. Natural languages allow ambiguity. Programming languages are designed to avoid ambiguity; an ambiguous program might have multiple meanings. Programming languages should precisely specify computations; ambiguity can destroy that precision.

Programming languages are, in general, designed to allow humans to express computations as sequences of operations. Computer processors, hereafter referred to as processors, microprocessors, or machines, are designed to execute sequences of operations. The operations that a processor implements are, for the most part, at a much lower level of abstraction than those specified in a programming language. For example, a programming language typically includes a concise way to print some number to a file. That single programming language statement translates into hundreds or thousands of machine operations that run when it executes.

The tool that performs such translations is called a compiler. The compiler takes as input a program written in some language and produces as its output an equivalent program. In the classic notion of a compiler, the output program is expressed in the operations available on some specific processor, often called the target machine. Viewed as an opaque box, a compiler might look like this:



Typical "source" languages might be C, C++, JAVA, ML, or RUBY. The "target" language is usually the instruction set of some processor.

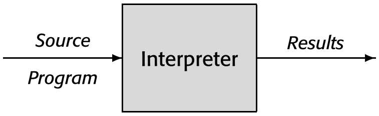
Instruction set
The set of operations that a processor provides is its *instruction set*.

The overall design of an instruction set is often called an *instruction set architecture* or ISA.

Some compilers produce a target program written in a human-oriented programming language rather than the assembly language of some computer. The programs that these compilers produce require further translation before they can execute directly on a computer. Many research compilers produce C programs as their output. Because C compilers are available on most computers, this makes the target program executable on all those systems, at the cost of an extra compilation for the final target. Compilers that target programming languages rather than the instruction set of a computer are often called *source-to-source translators*.

Many other systems qualify as compilers. For example, a typesetting program that produces PostScript can be considered a compiler. It takes as input a specification for how the document should look on the printed page and it produces as output a PostScript file. PostScript is simply a language for describing images. Because the typesetting program takes an executable specification and produces another executable specification, it is a compiler.

The code that turns PostScript into pixels is typically an *interpreter*, not a compiler. An interpreter takes as input an executable specification and produces as output the result of executing the specification.



Some languages, such as PERL, PYTHON, RUST, and SCHEME, are more often implemented with interpreters than with compilers.

Ahead-of-time compiler
a traditional compiler where translation occurs in a separate step from execution

Just-in-time compiler
a compiler where translation occurs at runtime

Historically, most compilers were designed to run in a separate step before execution. There were exceptions, but the *ahead-of-time* (AOT) compilation model was the rule. In recent years, programming languages and systems have emerged where it makes sense to delay compilation. These systems, called runtime optimizers or *just-in-time* (JIT) *compilers*, translate the code to executable form at runtime. This strategy adds the JIT-compilation cost to the program's running time, so the improvement in performance must compensate for the time spent in the JIT compiler.

Virtual machine
A virtual machine is a simulator for the ISA of an idealized processor.

JIT compilation can lead to complex translation schemes. For example, JAVA's execution model includes both compilation and interpretation. JAVA is compiled from source code into a more compact representation called JAVA *bytecode* by an AOT compiler. The application executes by running the bytecode on the JAVA Virtual Machine (JVM), an emulator, or interpreter, for JAVA bytecode. To improve performance, many implementations of the JVM include a JIT that compiles and optimizes heavily used bytecode sequences. Chapter 14 explores some of the issues that arise in building a JIT.

Interpreters and compilers have much in common. They perform many of the same tasks. Both analyze the input program and determine whether or not it is a valid program. Both build an internal model of the structure and meaning of the program. Both determine where to store values during execution. However, interpreting the code to produce a result is quite different from emitting a translated program that can be executed to produce the result. This book focuses on the problems that arise in building compilers. However, an implementor of interpreters may find much of the material relevant.

Why Study Compiler Construction?

The authors, of course, feel that the material in a course on compiler construction is intrinsically fascinating. Students, on the other hand, often want more concrete reasons to invest their time. We see two distinct answers: one pedagogical and the other practical.

For the student, compiler construction is a capstone exercise that brings together elements from across computer science and applies them in a large design and implementation project. A good compiler makes practical use of greedy algorithms (register allocation), heuristic search techniques (list scheduling), graph algorithms (dead-code elimination), dynamic programming (instruction selection), automata theory (scanning and parsing), and fixed-point algorithms (data-flow analysis). It deals with problems such as dynamic allocation, synchronization, naming, locality, memory hierarchy

MAY YOU STUDY COMPILERS IN INTERESTING TIMES

In 1951, Grace Murray Hopper wrote A0, considered by some to be the first compiler. By the early 1970s, compilers could generate high-quality code for the machines of the day. Why, then, is compiler construction still a subject of study and research? Changes in languages and computing environments pose a variety of challenges that require compiler writers to adapt their practices to obtain best results.

The general freeze on single-thread performance, decried as the end of Dennard scaling and Moore's law, has driven system designers to focus on improvements in architecture and parallelism. These changes, in turn, create new challenges in optimization and code generation.

The adoption of language features such as object orientation, late binding of names to types and classes, dynamic loading, and polymorphism has introduced new runtime costs. These features create new opportunities for program analyses and optimizations that reduce their runtime overhead.

Finally, compiled code has found application in new settings, from smartphones to hearing aids and doorbells, and new modes of use, from JIT compilation to shared code caches. As computing moves in new directions, the objectives and constraints for compilation change, bringing about new problems and new solutions.

management, and pipeline scheduling. Few other software systems bring together as many complex and diverse components.

Compilers demonstrate the successful application of theory to practice. The tools developed to generate scanners and parsers apply results from formal language theory. These same tools are used in text searches, website filters, and word processors; they are used to understand markup languages, scripting languages, and router specifications. Type checking and static analysis apply results from lattice theory, number theory, and other branches of mathematics to understand and improve programs. Code generators use algorithms for tree-pattern matching, parsing, dynamic programming, and text matching to automate the selection of instructions.

Building a successful compiler requires expertise, engineering, and planning. Good compilers approximate the solutions to hard problems. They emphasize efficiency, in their own implementations and in the code they generate. They have internal data structures and knowledge representations that expose the right level of detail—enough to allow strong optimization, but not enough to force the compiler to wallow in detail. Compiler construction brings together ideas and techniques from across the breadth of computer science and applies them in a constrained setting to solve some truly hard problems.

For the programmer, compiler construction reveals the cost of the abstractions that programs use, from case statements to procedure calls, from regex libraries to hash maps. A programmer cannot effectively tune an application's performance without knowing the costs of its individual parts. In general, programmers should design with appropriate abstractions. Then, if performance is an issue, they should measure performance, determine where the code spends its time, and reimplement that specific functionality in a lower-cost way.

This approach, however, requires that the programmer understands how the source-language's abstractions translate into the target-machine's code. For example:

- Equality comparisons between strings typically require time proportional to the length of the strings, while equality tests between integers are both $\mathsf{O}(1)$ and fast. This fact suggests hashing strings into small integers when possible.
- Consider implementing a simple translation $a = f(x)$ with a hash map versus a simple vector. Both are, arguably, $\mathsf{O}(1)$. The cost difference, however, might be a factor of ten or more. Whether or not this matters depends on the sparsity of *f*'s range and how often the code invokes *f*.

The study of compilation helps a programmer to understand many of the fundamental tradeoffs that arise in actual implementations. This knowledge should, in turn, inform better design decisions.

The Fundamental Principles of Compilation

Compilers are large, complex, carefully engineered objects. While many issues in compiler design are amenable to multiple solutions and interpretations, there are two fundamental principles that a compiler writer must keep in mind. The first principle is inviolable:

> *The compiler must preserve the meaning of the input program.*

Correctness is a fundamental issue in programming. The compiler must faithfully implement the "meaning" of its input program. This principle lies at the heart of the social contract between the compiler writer and the compiler user. If the compiler can take liberties with meaning, then why not simply generate a `nop` or a `return`? If an incorrect translation is acceptable, why expend the effort to get it right?

The second principle that a compiler must observe is practical:

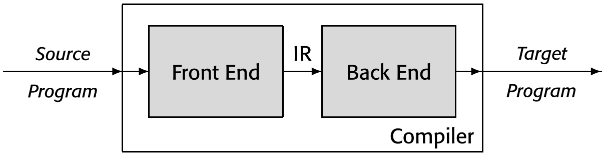
> *The compiler must discernibly improve the input program.*

A traditional compiler improves the input program by making it directly executable on some target machine. Other "compilers" improve their input in different ways. For example, `tpic` is a program that takes the specification for a drawing written in the graphics language `pic` and converts it into LATEX; the "improvement" lies in LATEX's greater availability and generality. A source-to-source translator that maps PYTHON into C should produce an output program that produces the same output as the original program; that output program should execute much faster than the original PYTHON code.

1.2 COMPILER STRUCTURE

A compiler is a large, complex software system. The community has been building compilers since 1951 and has learned many lessons about how to structure a compiler. In reality, a compiler is more complex than the simple opaque box shown earlier.

Fundamentally, a compiler must both understand the source program that it takes as input and map its functionality to the target machine. The distinct nature of these two tasks suggests a division of labor and leads to a design that decomposes compilation into two major pieces: a *front end* and a *back end*.



The front end focuses on understanding the source-language program. The back end focuses on mapping programs to the target machine. This separation of concerns has several important implications for the design and implementation of compilers.

Intermediate representation
A compiler uses some set of data structures to represent the code that it processes. That form is called an *intermediate representation*, or IR.

The front end must encode its knowledge of the source program in some structure, an *intermediate representation* (IR), for later use. The IR becomes the compiler's definitive representation for the code it is compiling. At each point in the process, the compiler will have a definitive representation. It may, in fact, use several different IRs for different purposes, but, at each point, one representation will be the definitive IR. We think of the definitive

A FEW WORDS ABOUT TIME

One of the hardest issues that arises in the study of compiler construction is *time*—specifically, keeping track of when various things happen. Some decisions are made when the compiler is designed, at *design time*. Some algorithms run when the compiler is built, at *build time*. Many activities take place when the compiler itself runs, at *compile time*. Finally, the compiled code can execute multiple times, at *runtime*.

The study of compiler construction involves thinking about each of these different times: what algorithms and data structures exist at each time, and how behavior that occurs at one time is planned at another time.

Each chapter, after this introduction, has a brief discussion of time as part of the introduction. These discussions, under the header *A Few Words About Time*, highlight how the techniques described in that chapter fit into these different temporal slots.

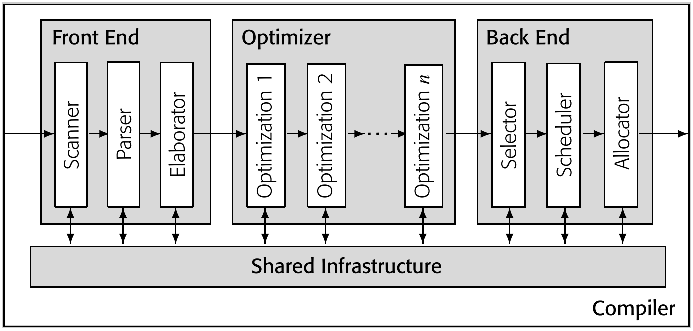
IR as the version of the program passed between independent phases of the compiler, like the IR passed from the front end to the back end in the preceding drawing.

In a two-phase compiler, the front end must ensure that the source program is well formed, and it must map that code into the IR. The back end must map the IR program into the instruction set and the finite resources of the target machine. Because the back end only processes an IR created by the front end, it can assume that the IR contains no syntactic or semantic errors.

The compiler can make multiple passes over the IR form of the code before emitting the target program. Because the compiler can analyze the code in one phase and use the resulting knowledge in another, this multipass approach leads to better code than a single pass can produce. This strategy requires that knowledge derived in the first pass be recorded in the IR, where later passes can find and use it.

Retargeting
The task of changing the compiler to generate code for a new processor is often called *retargeting* the compiler.

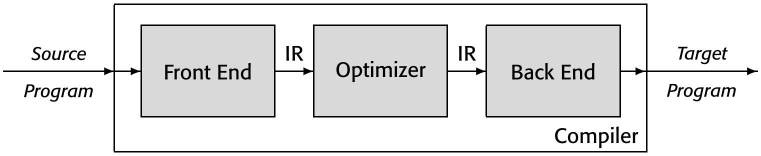
Finally, the two-phase structure may simplify the process of *retargeting* the compiler. We can easily envision constructing multiple back ends for a single front end to produce compilers that accept the same language but target different ISAs. Similarly, we can envision front ends for different languages producing the same IR and using a common back end. Both scenarios assume that one IR can serve for several combinations of source and target; in practice, both language-specific and machine-specific details usually find their way into the IR.



■ **FIGURE 1.1** Internal Structure of a Typical Compiler.

Introducing an IR lets the compiler writer add a phase between the front end and the back end. This middle phase, or *optimizer*, tries to improve the IR program. With the IR as an interface, the compiler writer can insert optimizations with minimal disruption to the front end and back end. The result is a three-phase compiler.

Optimizer
The middle section of a compiler, called an *optimizer*, analyzes and transforms the IR in an attempt to improve it.



The optimizer is an IR-to-IR transformer, or a series of them, that tries to improve the IR program in some way. The optimizer can make one or more passes over the IR, analyze the IR, and rewrite the IR. The optimizer may rewrite the IR in a way that is likely to produce a faster target program from the back end or a smaller target program from the back end. It may have other objectives, such as a program that produces fewer page faults or uses less energy.

The individual passes, or transformations, in an optimizer are, themselves, compilers according to our earlier definition.

Conceptually, the three-phase structure represents the classic optimizing compiler. In practice, each phase is divided internally into a series of passes. The front end has two or three distinct passes that combine to recognize valid source-language programs and produce the initial IR form of the program. The optimizer contains passes that use distinct analyses and transformations to improve the code. The number and purpose of these passes vary from compiler to compiler. The back end consists of a series of passes, each of which takes the IR program one step closer to the target machine's instruction set. The three phases and their individual passes share a common infrastructure. This structure is shown in Fig. 1.1.

SEPARATE COMPILATION

Most compiler systems support *separate compilation*. The programmer can compile distinct files of code in independent steps, often at different times. Each of these compilations produces a file of object code. Multiple object code files can be linked together to form a single executable.

Separate compilation lets the programmer limit the amount of code that must be compiled after a change to the program. It saves a huge amount of compile time. (Imagine changing one line of code in a million-line program and having to recompile the entire program.) It enables the construction and use of libraries. It allows multiple programmers to work independently on a single large application. Separate compilation is critical to our ability to build software at scale.

In practice, the conceptual division of a compiler into three phases, a front end, an optimizer, and a back end, creates a useful separation of concerns. Each phase addresses a different set of problems. The front end works to understand the source program and record the results of that analysis in IR form. The optimizer tries to improve the IR so that it produces a more efficient execution. The back end maps the optimized IR program onto the bounded resources of the target machine's ISA in a way that makes efficient use of resources.

Building a compiler as a sequence of independent passes that communicate by using one or more common IRs has another significant advantage. This structure lends itself more readily to debugging than does a monolithic compiler. During development, the compiler writer can insert a tool between passes that checks the validity of the IR. The compiler writer can remove passes and test them independently, or test subsets to look for unplanned and unexpected interactions.

Of these three phases, the optimizer has the murkiest description. The term *optimization* implies that the compiler discovers an optimal solution to some problem. The issues and problems that arise in code optimization are so complex and interrelated that they cannot, in practice, be solved optimally. Furthermore, the actual behavior of the compiled code depends on interactions among all of the techniques applied in the compiler. Thus, even if one pass can be proved optimal, its interactions with other passes may produce less than optimal results. As a result, a good optimizing compiler can improve the quality of the code, relative to an unoptimized version. However, an optimizing compiler makes no guarantee of optimality.

The optimizer can be a single monolithic pass that applies one or more optimizations to improve the code, or it can be structured as a series of smaller passes with each pass reading and writing IR. The monolithic structure may be more efficient. The multipass structure may lend itself to a less complex implementation and a simpler approach to debugging the compiler. It also creates the flexibility to employ different sets of optimizations in different situations. The choice between these two approaches depends on the constraints under which the compiler is built and operates.

1.3 OVERVIEW OF TRANSLATION

To translate code written in a programming language into code that runs on the target machine, a compiler runs through many steps. To make this abstract process more concrete, consider the steps needed to generate executable code for the following expression:

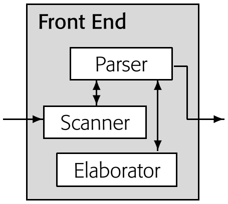
```
a ← a × 2 × b × c × d
```

where `a`, `b`, `c`, and `d` are variables, ← indicates an assignment, and × is the multiply operator. The following subsections trace the path that a compiler takes to turn this simple assignment into executable code.

1.3.1 The Front End

Before the compiler can translate an expression into executable target-machine code, it must understand both its form, or *syntax*, and its meaning, or *semantics*. The front end determines if the input code is well formed, in terms of both syntax and semantics. If it finds that the code is valid, it creates a representation of the code in the compiler's IR; if not, it reports back to the user with diagnostic error messages to identify the problems with the code.

Fig. 1.1 shows the classic view of a compiler, as a series of passes. While the optimizer and the back end work this way, the front end actually has a somewhat different control structure, as shown in the margin. The scanner converts the stream of characters from the input code into a stream of words. It recognizes valid words by their spellings; for example, in most programming languages, `1g2h3i` is neither a valid identifier nor a valid number. The parser fits the words from the scanner to a rule-based model of the input language's syntax, called a *grammar*. It calls the scanner incrementally as it needs additional words. As the parser matches words to rules in the grammar, it may call on the elaborator to perform additional computation on the input program. Examples of such computation include building an IR, checking type consistency, or laying out storage.



Grammar
A model of the syntax of a programming language

NOTATION

Compiler books are, in essence, about notation. After all, a compiler translates a program written in one notation into an equivalent program written in another notation. A number of notational issues will arise in your reading of this book. In some cases, these issues will directly affect your understanding of the material.

Expressing Algorithms We have tried to keep the algorithms concise. They are written at a relatively high level, assuming that the reader can supply implementation details. They are written in a *slanted, sans-serif font*. Indentation is both deliberate and significant, as in PYTHON. Indented code forms a block. In the following code fragment,

```
if Action[s,word] = "shift $s_i$" then
    push $\langle word, s_i \rangle$
    word ← NextWord( )
else if ···
```

all the statements between the `then` and the `else` are part of the `then` clause of the `if–then–else` construct. In the case of a single statement block, we sometimes write it inline with the `then` or `else` when the meaning is clear.

Writing Code In some examples, we show actual program text written in some language chosen to demonstrate a particular point. Actual program text is written in a `monospace` font.

Arithmetic Operators Finally, we have forsaken the traditional use of * for × and of / for ÷, except in actual program text. The meaning should be clear to the reader.

Scanners and parsers find widespread application in other systems. Tools generate online content in markup languages, such as HTML; browsers scan and parse that content to display it. To exchange data, systems rewrite it into an external format, such as YAML or standard CSV format; programs that read the data must scan and parse it. A spreadsheet application knows that a function, such as `average`, takes an argument list; it must scan and parse formulas to find function names and to ensure the correctness of the argument lists.

Scanners and parsers are embedded in myriad other tools. Some of them are implemented with generators, as described in Chapters 2 and 3. Others are implemented with ad-hoc techniques. Nonetheless, all of these scanners and parsers should look familiar to a compiler writer.

Checking Syntax

To check the syntax of the input program, the compiler must compare the program's structure against the grammar that defines the language. The grammar provides a formal definition of the language's syntax. It also leads to an efficient mechanism to test whether the input forms a sentence in the language.

A grammar defines a set, usually infinite, of strings of words. It has a set of rules, called *productions*, defined over the words in the language and a set of syntactic variables introduced to provide structure.

In a grammar, the rules use parts of speech, or syntactic categories, rather than individual words. Thus, a single rule represents many sentences. For example, many English sentences have the form:

$$\mathit{Sentence} \rightarrow \mathit{Subject}\ \ \texttt{verb}\ \ \mathit{Object}\ \ \texttt{endmark}$$

where `verb` and `endmark` are parts of speech, and *Sentence*, *Subject*, and *Object* are syntactic variables. *Sentence* represents any string with the form described by this rule. The symbol "→" reads "derives" and means that an instance of the right-hand side can be abstracted to the syntactic variable on the left-hand side.

Scanner
the compiler pass that converts a string of characters into a stream of classified words

Consider a sentence like "Compilers are engineered systems." The first step in understanding the syntax of this sentence is to identify distinct words in the sentence and to classify each word according to its part of speech. In a compiler, the scanner performs this function. The scanner converts the input, a stream of characters, into a stream of classified words—that is, pairs of the form (p, s), where p is the word's *part of speech* and s is its spelling. A scanner might convert the example sentence into the following stream of classified words:

(`noun`, "Compilers"), (`verb`, "are"), (`adjective`, "engineered"),
(`noun`, "systems"), (`endmark`, ".")

Equality tests on strings take time proportional to string length. Equality tests on small integers are usually $\mathbf{O}(1)$.

In practice, the string with the actual spelling of a word might be stored in a hash table and represented in the pair with an integer index to simplify equality tests. Chapter 2 explores the theory and practice of scanner construction.

In the next step, the compiler tries to match the stream of words to the rules of the grammar. For example, a working knowledge of English might include the grammatical rules shown in Fig. 1.2. By inspection, we can

1	*Sentence*	→	*Subject* verb *Object* endmark
2	*Subject*	→	noun
3	*Subject*	→	*Modifier* noun
4	*Object*	→	noun
5	*Object*	→	*Modifier* noun
6	*Modifier*	→	adjective

■ **FIGURE 1.2** Simple English Rules.

discover the following *derivation* for the sentence "Compilers are engineered systems."

Rule	Prototype Sentence
—	*Sentence*
1	*Subject* verb *Object* endmark
2	noun verb *Object* endmark
5	noun verb *Modifier* noun endmark
6	noun verb adjective noun endmark

The derivation starts with the syntactic variable *Sentence*. At each step, it rewrites one syntactic variable in the prototype sentence, replacing it with a right-hand side that can be derived from that rule. The first step uses rule 1 to replace *Sentence*. The second step uses rule 2 to replace *Subject*. The third step replaces *Object* using rule 5, while the final step rewrites *Modifier* with adjective according to rule 6. At this point, the derived sentence matches the stream of categorized words produced by the scanner.

Parser
the compiler pass that determines if the input stream is a sentence in the source language

The derivation proves that the sentence "Compilers are engineered objects." belongs to the language described by the simple grammar formed by rules 1 through 6. The sentence is grammatically correct. The process of automatically finding derivations is called *parsing*. Chapter 3 presents the techniques that compilers use to parse the input program.

A grammatically correct sentence can be meaningless. For example, the sentence "Rocks are green vegetables." has the same parts of speech in the same order as "Compilers are engineered objects." but has no rational meaning. To understand the difference between these two sentences requires contextual knowledge about software systems, rocks, and vegetables.

When the parser finds a derivation, thus proving that the input program is syntactically correct, it must create the various data structures required for the rest of the compiler. That process includes building an IR version of the input program, augmented with tables that describe each of the names in the program; using that IR version of the code to check for type consistency; and assigning storage to every value that the program will compute. Some of these computations, such as building the IR for the input, are performed incrementally as the parser works. Others, such as storage assignment, are accomplished by iterating over the IR after the parse completes.

We refer to these additional computations as semantic elaborations. They expand the "meaning" of the program from simple syntax to an operational definition. During these computations, the compiler may detect errors that occur at a deeper level than the grammar can specify. Examples include type mismatches and errors of number, such as an array reference with the wrong number of dimensions or a procedure call with too many parameters. Chapter 5 explores some of the issues that arise in semantic elaboration and the mechanisms that compiler writers use to address them.

Intermediate Representations

The final issue handled in the front end of a compiler is the generation of an IR form of the code. A typical IR consists of a representation of the code, plus data structures that contain additional information.

Compilers use a variety of kinds of IR, depending on the source language, the target language, and the specific goals of the compiler. Some IRs represent the program as a graph. Other IRs resemble a sequential assembly code program. The code in the margin shows how our example expression, $a \leftarrow a \times 2 \times b \times c \times d$, might look in a low-level, sequential IR. Chapter 4 presents an overview of the kinds of IRs that compilers use.

```
$t_0$ ← a × 2
$t_1$ ← $t_0$ × b
$t_2$ ← $t_1$ × c
$t_3$ ← $t_2$ × d
a     ← $t_3$
```

Low-level, Sequential IR
for $a \leftarrow a \times 2 \times b \times c \times d$

For each source-language construct, the compiler needs a strategy for how it will implement that construct in the IR. Specific choices affect the compiler's ability to transform and improve the code. We spend both Chapter 4 and Chapter 5 on the issues that arise in generation of IR for source-code constructs. Procedure calls are, at once, both a source of inefficiency in the final code and the fundamental glue that pieces together different source files into a complete program. Chapter 6 focuses on the implementation of procedure calls. Chapter 7 then presents implementation strategies for most other programming language constructs.

1.3.2 **The Optimizer**

When the front end emits IR for the input program, it handles the statements one at a time, in the order that they are encountered. Thus, the initial IR program contains general implementation strategies that will work in any surrounding context that the compiler might generate. At runtime, the code

ABOUT ILOC

Throughout the book, low-level examples are written in an IR named ILOC—an acronym derived from "intermediate language for an optimizing compiler." Over the years, ILOC has undergone many changes. Appendix A describes ILOC in detail.

Think of ILOC as the assembly language for a simple RISC machine. It has a standard set of operations. Most operations read their operands from registers and write their results to registers. The memory operations, `load` and `store`, transfer values between memory and registers. To simplify the exposition in this book, most examples assume that all data consists of integers.

Each operation has a set of operands and a set of results. The operation has five parts: an opcode, a list of operands, a separator, a list of results, and an optional comment. To store the sum of registers 1 and 2 into register 3, the programmer might write

add $r_1, r_2 \Rightarrow r_3$ // example instruction

The separator, ⇒, precedes the result list. It is a visual reminder that information flows from left to right. It disambiguates cases where a person reading the assembly-level text can easily confuse operands and targets. (See `loadAI` and `storeAI`, for example.)

To facilitate the discussion in Chapter 7, ILOC has multiple ways to write a conditional branch. It supports a Boolean compare and branch as well as a more traditional condition code–based branch. In either case branches always have labels for both the taken path and the "not-taken" path, which makes all interblock transitions explicit. ILOC does not support a "fall-through" case on a branch.

will execute in a more constrained and predictable context. The optimizer can analyze the IR form of the code, discover facts about that context, and use the resulting knowledge to rewrite the code so that it computes the same answer in a more efficient way.

Elapsed time
The elapsed time of a program refers to the amount of time that it would require to run standalone on a system.

Efficiency can have many meanings. Classic code optimization tries to reduce the application's elapsed running time. In other circumstances, the optimizer might try to reduce the size of the compiled code, or the energy that the processor consumes as the code runs. Compilers have targeted each of these different notions of efficiency. Chapter 8 provides an introduction to code optimization and looks at example techniques across a number of different-sized code regions.

```
b ← ···
c ← ···
a ← 1
for i = 1 to n
    read d
    a ← a × 2 × b × c × d
    end
```

(a) Original Code in Context

```
b ← ···
c ← ···
a ← 1
t ← 2 × b × c
for i = 1 to n
    read d
    a ← a × d × t
    end
```

(b) Improved Code

■ **FIGURE 1.3** Context Makes a Difference.

To make these ideas concrete, Fig. 1.3 shows our example assignment, a ← a × 2 × b × c × d, embedded inside a simple loop. The assignment uses five values: a, b, c, d, and 2. The loop updates a and d in every iteration. By contrast, b, c, and 2 do not change as the loop executes; they are *loop invariant*. If the optimizer can discover their invariance, it can rewrite the code as shown in Fig. 1.3(b). In this version of the code, the number of multiplications has been reduced from $4n$ to $2n + 2$. For $n > 1$, the rewritten loop should execute faster.

Loop invariant
A value that does not change between iterations of a loop is said to be *loop-invariant*.

Analysis

Most optimizations consist of an analysis and a transformation. The analysis determines where the compiler can safely and profitably apply the transformation. Compilers use several kinds of analysis to support transformations. *Data-flow analysis* reasons, at compile time, about the flow of values at runtime. Data-flow analyzers typically solve a system of simultaneous set equations based on facts derived from the IR form of the code. *Dependence analysis* uses number-theoretic tests to reason about the relative independence of memory references. Compilers use it to disambiguate array-element references. Chapter 9 explores data-flow analysis, along with the translation into and out of static-single-assignment form, a widely used IR that encodes information about the flow of both values and control directly in the IR.

Data-flow analysis
a form of compile-time reasoning about the runtime flow of values

Transformation

To improve the code, the compiler must go beyond analyzing it. The compiler must use the results of analysis to rewrite the code into a more efficient form. Myriad transformations have been invented to improve the time or space requirements of executable code. Some, such as the loop-invariant code motion shown in Fig. 1.3(b), improve the running time of the program. Others make the code more compact. Transformations vary in their effect, the scope over which they operate, and the analysis required to support them. The literature on transformations is rich; the subject is large enough and deep enough to merit one or more separate books. Chapter 10 explores scalar transformations—transformations intended to improve the performance of code on a single processor. It presents a taxonomy for organizing the subject and populates that taxonomy with examples.

1.3.3 The Back End

The compiler's back end traverses the IR and emits code for the target machine. The back end solves at least three major problems.

We distinguish carefully between an *operation*, a single atomic command executed by one processor or functional unit, and an *instruction*, the set of one or more operations that start to execute in the same cycle on a processor.

- It must convert the IR operations into equivalent operations in the target processor's ISA, a process called *instruction selection*, which is the subject of Chapter 11.
- It must select an execution order for the operations, a process called *instruction scheduling*, which is the subject of Chapter 12.
- It must decide which values should reside in registers at each point in the code, a process called *register allocation*, which is the subject of Chapter 13.

Most compilers handle these three processes separately. These three distinct but related processes are often lumped together in the term "code generation," even though the instruction selector has the primary responsibility for generating target-machine instructions.

Each of these three problems is, on its own, a computationally hard problem. While it is not clear how to define optimal instruction selection, the problem of generating the fastest code sequence for a procedure on a given ISA involves considering a huge number of alternatives. Instruction scheduling is NP-complete for a basic block under most realistic execution models; moving to larger regions of code does not simplify the problem. Register allocation is, in its general form, also NP-complete in procedures with control flow.

Instruction Selection

The first stage of code generation rewrites the IR operations into target machine operations, a process called *instruction selection*. Instruction selection maps each IR operation, in its context, into one or more target machine operations. To illustrate the process, consider rewriting our example expression, $a \leftarrow a \times 2 \times b \times c \times d$, into ILOC code. The low-level IR for the assignment appears in the margin.

$$\begin{array}{lll} t_0 & \leftarrow & a \times 2 \\ t_1 & \leftarrow & t_0 \times b \\ t_2 & \leftarrow & t_1 \times c \\ t_3 & \leftarrow & t_2 \times d \\ a & \leftarrow & t_3 \end{array}$$

Low-level IR for $a \leftarrow a \times 2 \times b \times c \times d$

```
loadAI   r_arp, @a ⇒ r_a        // load 'a'
loadI    2         ⇒ r_2        // constant 2 into r_2
loadAI   r_arp, @b ⇒ r_b        // load 'b'
loadAI   r_arp, @c ⇒ r_c        // load 'c'
loadAI   r_arp, @d ⇒ r_d        // load 'd'
mult     r_a, r_2  ⇒ r_a        // r_a ← a × 2
mult     r_a, r_b  ⇒ r_a        // r_a ← (a × 2) × b
mult     r_a, r_c  ⇒ r_a        // r_a ← (a × 2 × b) × c
mult     r_a, r_d  ⇒ r_a        // r_a ← (a × 2 × b × c) × d
storeAI  r_a       ⇒ r_arp, @a  // write r_a back to 'a'
```

■ **FIGURE 1.4** Example in ILOC.

Fig. 1.4 shows one ILOC sequence that the compiler might generate. It uses just four ILOC operations:

r_{arp}
the register designated to hold a pointer to the current activation record

The local data area begins at a fixed offset from the address in r_{arp}.

ILOC Operation			Meaning
loadAI	r_1, c_2	$\Rightarrow r_3$	Memory $(r_1 + c_2) \rightarrow r_3$
loadI	c_1	$\Rightarrow r_2$	$c_1 \rightarrow r_2$
mult	r_1, r_2	$\Rightarrow r_3$	$r_1 \times r_2 \rightarrow r_3$
storeAI	r_1	$\Rightarrow r_2, c_3$	$r_1 \rightarrow$ Memory $(r_2 + c_3)$

The code assumes that a, b, c, and d are located at offsets @a, @b, @c, and @d from the address contained in the register r_{arp}. The code loads all of the relevant values into registers, performs the multiplications in order, and stores the result to the memory location for a.

The code assumes an unlimited supply of *virtual registers*. It names them with symbolic names such as r_a to hold a and r_{arp} to hold the address where the data storage for our named values begins. The instruction selector relies on the fact that the register allocator will map the virtual registers into physical registers on the target machine.

Virtual register
a symbolic register name that the compiler uses to indicate a value that can be stored in a register

The instruction selector can take advantage of special operations on the target machine. For example, if an immediate-multiply operation, multI, is available, it might replace mult $r_a, r_2 \Rightarrow r_a$ with multI $r_a, 2 \Rightarrow r_a$, eliminating the need for the operation loadI $2 \Rightarrow r_2$ and reducing the demand for registers by one. Alternatively, if addition is faster than multiplication, it might simply replace the loadI – mult sequence with add $r_a, r_a \Rightarrow r_a$, which should be faster. Chapter 11 presents two techniques for instruction selection that use pattern matching to choose efficient implementations for IR operations.

Physical register
a name that represents an actual register in the target machine's ISA

Register Allocation

During instruction selection, the compiler deliberately ignores the fact that the target machine has a limited set of physical registers. Instead, it uses virtual registers and assumes that "enough" registers exist. In practice, the earlier stages of compilation may create demand for more physical registers than the hardware provides. The register allocator must map those virtual registers onto physical registers. Thus, the allocator must decide, at each point in the code, which values will reside in physical registers. It then rewrites the code to reflect its decisions. For example, a register allocator might minimize register use by rewriting the code in our example as follows:

```
loadAI   r_arp, @a ⇒ r_1          // load 'a'
add      r_1, r_1  ⇒ r_1          // r_1 ← a × 2
loadAI   r_arp, @b ⇒ r_2          // load 'b'
mult     r_1, r_2  ⇒ r_1          // r_1 ← (a × 2) × b
loadAI   r_arp, @c ⇒ r_2          // load 'c'
mult     r_1, r_2  ⇒ r_1          // r_1 ← (a × 2 × b) × c
loadAI   r_arp, @d ⇒ r_2          // load 'd'
mult     r_1, r_2  ⇒ r_1          // r_1 ← (a × 2 × b × c) × d
storeAI  r_1       ⇒ r_arp, @a    // write r_1 back to 'a'
```

This sequence uses three registers instead of six.

Minimizing register use may be counterproductive. If, for example, any of the named values, a, b, c, or d, are already in registers, the code should reference those registers directly. If all are in registers, the sequence could be implemented so that it required no additional registers. Alternatively, if some nearby expression also computed a × 2, it might be better to preserve that value in a register rather than to recompute it later. This optimization eliminates a later evaluation of a × 2, at the cost of an increase in demand for registers.

Liveness
A value *v* is *live* at some point *p* if there exists a path from *p* to a use of *v* along which *v* is not redefined.

To manage this kind of tradeoff, the register allocator analyzes the code to determine where each value, typically represented as a virtual register, is *live*. The allocator's goal is to keep each value in a register in the range of code where that value is live. When that is not possible, it should choose the set of enregistered values in a way that tries to minimize the runtime cost introduced by keeping some of those values in memory. Chapter 13 explores the problems that arise in register allocation and the techniques that compiler writers use to solve them.

Instruction Scheduling

To produce code that executes quickly, the code generator may need to reorder operations to reflect the target machine's specific performance constraints. The execution time of the different operations can vary. Memory access operations can take tens or hundreds of cycles, while some arithmetic operations, particularly multiplication and division, take several cycles. The impact of these longer latency operations on the performance of compiled code can be dramatic.

Assume, for the moment, that a `loadAI` or `storeAI` operation requires three cycles, a `mult` requires two cycles, and all other operations require one cycle. The following table shows how the previous code fragment performs under these assumptions. The **Start** column shows the cycle in which each operation begins execution and the **End** column shows the cycle in which it completes.

Elapsed time (again)
For a block of straight-line code, the elapsed time is simply the total number of processor cycles required to complete the block, multiplied by the time per cycle.

Start	End	Code
1	3	loadAI r_{arp}, @a ⇒ r_1 // load 'a'
4	4	add r_1, r_1 ⇒ r_1 // $r_1 \leftarrow$ a × 2
5	7	loadAI r_{arp}, @b ⇒ r_2 // load 'b'
8	9	mult r_1, r_2 ⇒ r_1 // $r_1 \leftarrow$ (a × 2) × b
10	12	loadAI r_{arp}, @c ⇒ r_2 // load 'c'
13	14	mult r_1, r_2 ⇒ r_1 // $r_1 \leftarrow$ (a × 2 × b) × c
15	17	loadAI r_{arp}, @d ⇒ r_2 // load 'd'
18	19	mult r_1, r_2 ⇒ r_1 // $r_1 \leftarrow$ (a × 2 × b × c) × d
20	22	storeAI r_1 ⇒ r_{arp}, @a // write r_1 back to 'a'

This nine-operation sequence takes 22 cycles to execute. Minimizing register use did not lead to rapid execution.

Many processors have the property that they can initiate new operations while a long-latency operation executes. As long as the results of a long-latency operation are not referenced until the operation completes, execution proceeds normally. If, however, some intervening operation tries to read the result of the long-latency operation prematurely, the processor delays the operation that needs the value until the long-latency operation completes. An operation cannot begin to execute until its operands are ready, and its results are not ready until the operation terminates.

The instruction scheduler reorders the operations in the code. It attempts to minimize the number of cycles wasted waiting for operands. Of course, the scheduler must ensure that the new sequence produces the same result as the original sequence. In many cases, the scheduler can drastically improve the performance of “naive” code. A scheduler might produce the following sequence for our example:

Start	End	Code
1	3	loadAI r_{arp}, @a ⇒ r_1 // load 'a'
2	4	loadAI r_{arp}, @b ⇒ r_2 // load 'b'
3	5	loadAI r_{arp}, @c ⇒ r_3 // load 'c'
4	4	add r_1, r_1 ⇒ r_1 // r_1 ← a × 2
5	6	mult r_1, r_2 ⇒ r_1 // r_1 ← (a × 2) × b
6	8	loadAI r_{arp}, @d ⇒ r_2 // load 'd'
7	8	mult r_1, r_3 ⇒ r_1 // r_1 ← (a × 2 × b) × c
9	10	mult r_1, r_2 ⇒ r_1 // r_1 ← (a × 2 × b × c) × d
11	13	storeAI r_1 ⇒ r_{arp}, @a // write r_1 back to 'a'

This version of the code requires just 13 cycles to execute. The code uses one more register than the minimal number. It starts an operation in every cycle except 8, 10, and 12. Other equivalent schedules are possible, as are equal-length schedules that use more registers. Chapter 12 presents several scheduling techniques that are widely used.

Interactions Among Code-Generation Components

Many of the truly hard problems that occur in compilation arise during code generation. To make matters more complex, these problems interact. For example, instruction scheduling moves `load` operations away from the operations that use the newly loaded values. This can increase the period over which the values are needed and, correspondingly, increase the demand for registers in that interval in the code. Similarly, the assignment of particular values to specific registers can constrain instruction scheduling by creating a “false” dependence between two operations. (The second operation cannot be scheduled until the first completes, even though the values in the common register are independent.) Renaming the values can often eliminate this false dependence, at the potential cost of using more registers.

1.4 ENGINEERING

Compiler construction is an exercise in engineering—design and implementation under constraints. A typical compiler has a series of passes that, together, translate code from some source language into some target language. This structure creates a distinct separation of concerns. The parser does not pay attention to the way registers are used and the register allocator does not worry about whether or not some constant "13" has a leading zero. The pass structure of modern compilers simplifies implementation, debugging, and maintenance.

A compiler uses dozens of algorithms and data structures. The compiler writer must select, for each step in the translation, one or more efficient and effective techniques. This design process involves tradeoffs in running time, in implementation complexity, and in effectiveness. For any particular problem, the compiler writer can find multiple distinct solutions; each of them will have its own strengths, weaknesses, costs, and benefits. The compiler writer weaves together a set of solutions that, in concert, produce the desired result.

A successful compiler executes an unimaginable number of times. Consider the total number of times that the GCC compiler has run. Over GCC's lifetime, even small inefficiencies add up to a significant amount of time. The savings from good design and implementation accumulate over time. Thus, compiler writers must pay attention to compile time costs, such as the asymptotic complexity of algorithms, the actual running time of the implementation, and the space used by data structures. The compiler writer should have in mind a budget for how much time the compiler will spend on its various tasks.

The compiler's front end should be fast. The theory-based techniques presented in Chapters 2 and 3 lead to scanners and parsers that operate in $\mathsf{O}(n)$ time, where n is a measure of the size and structure of the input program. By contrast, optimization and code generation address problems that require more time. Solutions to many of these problems use algorithms with complexities greater than $\mathsf{O}(n)$. Thus, algorithm choice in the optimizer and back end has a larger impact on compile time than it does in the front end. The compiler writer may need to trade precision of analysis and effectiveness of optimization against increases in compile time. Design and implementation decisions should be made consciously and carefully.

JIT construction may be the ultimate feat of compiler engineering. Because the system must recoup compile time through improved running time, the JIT writer faces more extreme constraints than the author of a traditional compiler. Chapter 14 explores the tradeoffs that arise in the design and implementation of runtime optimizers.

1.5 SUMMARY AND PERSPECTIVE

Compiler construction is a complex task. A good compiler combines ideas from formal language theory, from the study of algorithms, from artificial intelligence, from systems design, from computer architecture, and from the theory of programming languages and applies them to the problem of translating a program. A compiler brings together greedy algorithms, heuristic techniques, graph algorithms, dynamic programming, automata, fixed-point algorithms, synchronization and locality, allocation, and naming. Many of the problems that confront the compiler are too hard or computationally intensive for it to solve optimally; thus, compilers use approximations, heuristics, and rules of thumb. As a result, compilers contain complex interactions that can lead to surprising results—both good and bad.

To place this activity in an orderly framework and to provide for careful separation of concerns, most compilers are organized into three major phases: a front end, an optimizer, and a back end. Each phase has a different set of problems to tackle, and the approaches used to solve those problems differ, too. The front end focuses on translating source code into some IR. Front ends rely on results from formal language theory and type theory, with a healthy dose of algorithms and data structures. The middle section, or optimizer, translates one IR program into another, with the goal of producing an IR program that executes efficiently. Optimizers analyze programs to derive knowledge about their runtime behavior and then use that knowledge to transform the code and improve its behavior. The back end maps an IR program to the instruction set of a specific processor. A back end approximates the answers to hard problems in allocation and scheduling, and the quality of its approximation has a direct impact on the speed and size of the compiled code.

The material in Chapters 5 through 7 is heavily interrelated. We recommend that you read all three chapters, then reread them a second time.

This book explores each of these phases. Chapters 2 and 3 describe the algorithms used in a compiler's front end. Chapter 4 provides an introduction to IRs. Chapters 5 through 7 explore the implementation of various programming language abstractions, including procedures and control structures, along with the mechanisms used to create them in IR form. Chapter 8 provides an introduction to code optimization. Chapters 9 and 10 provide a more detailed treatment of analysis and optimization for the interested reader. Chapters 11 through 13 cover the techniques used by back ends for instruction selection, scheduling, and register allocation. Finally, Chapter 14 describes how all of these techniques come together in modern runtime optimizers or just-in-time compilers.

CHAPTER NOTES

The first compilers appeared in the 1950s. These early systems showed surprising sophistication. The original FORTRAN compiler was a multipass system that included a distinct scanner, parser, and register allocator, along with some optimizations [27,28]. The Alpha system, built by Ershov and his colleagues, performed local optimization [150] and used graph coloring to reduce the amount of memory needed for data items [151,152]. Early ALGOL-60 efforts developed many critical ideas that still play important roles in compiler implementation; see the January 1961 issue of Communications of the ACM (CACM) for examples [160,213,214,316].

Knuth provides some interesting recollections of compiler construction in the early 1960s [238]. Randell and Russell describe early implementation efforts for ALGOL-60 [303]. Allen describes the history of compiler development inside IBM with an emphasis on the interplay of theory and practice [15].

Many influential compilers were built in the 1960s and 1970s. These include the classic optimizing compiler FORTRAN H [260,317], the Bliss-11 and Bliss-32 compilers [79,368], and the portable BCPL compiler [308]. These compilers produced high-quality code for a variety of CISC machines. Compilers for students, on the other hand, focused on rapid compilation, good diagnostic messages, and error correction [104,156].

The advent of RISC architecture in the 1980s led to another generation of compilers; these focused on strong optimization and code generation [25,88, 96,215]. These compilers featured full-blown multipass optimizers. Modern RISC compilers still follow this model.

During the 1990s, compiler-construction research focused on reacting to the rapid changes taking place in microprocessor and system architecture. The decade began with Intel's *i*860 processor challenging compiler writers to manage pipelines and memory latencies directly. At its end, compilers confronted challenges that ranged from multiple functional units to long memory latencies to parallel code generation. The structure and organization of 1980s RISC compilers proved flexible enough for these new challenges. Researchers simply added new passes to their optimizers and code generators.

While JAVA systems use a mix of compilation and interpretation [69,288], JAVA is not the first language to employ such a mix. LISP systems have long included both native-code compilers and virtual-machine implementation schemes [275,336]. The SMALLTALK-80 system used a bytecode distribution and a virtual machine [242]; several implementations added JIT compilers [137].

In the post-2000 years, declines in the cost of computation let compiler writers budget more cycles for analysis and transformation. This effect led to the adoption of more powerful analyses for problems, including whole program and library analysis [372], and widespread use of pointer analysis to disambiguate memory references [203]. Rising memory latencies increased the importance of locality-improving and latency-hiding optimizations [2,52,350].

EXERCISES

1. A compiler writer faces many tradeoffs. What are the five qualities that you, as a user, consider most important in a compiler that you purchase? Does that list change when you are the compiler writer? What does your list tell you about a compiler that you would implement?
2. Consider a simple web browser that takes as input a textual string in HTML format and displays the specified graphics on the screen. Is the display process one of compilation or interpretation?
3. Compilers are used in many different circumstances. What differences might you expect in compilers designed for the following applications?

   a. A just-in-time compiler used to translate user interface code downloaded over a network

   b. A compiler that targets the embedded processor used in a mobile device

   c. A compiler used in an introductory programming course at a high school

   d. A compiler used to build wind-tunnel simulations that run on a large cluster of identical processors

   e. A compiler that targets numerically intensive programs to a large number of diverse machines

Chapter 2

Scanners

ABSTRACT

The scanner's task is to transform a stream of characters into a stream of words in the input language. Each word must be classified into a syntactic category, or "part of speech." The scanner is the only pass in the compiler to touch every character in the input program. Compiler writers place a premium on speed in scanning, in part, because the scanner's input is larger, in some measure, than that of any other pass, and, in part, because highly efficient techniques for scanning are easy to understand and to implement.

This chapter introduces regular expressions, a notation used to describe the set of valid words in a programming language. It develops the formal mechanisms to generate scanners from regular expressions, either manually or automatically.

KEYWORDS

Scanner, Finite Automaton, Regular Expression, Fixed-Point Algorithm

2.1 INTRODUCTION

Scanning is the first stage of the three-part process that the compiler's front end uses to understand the input program. The scanner, or lexical analyzer, reads a stream of characters and produces a stream of words. The parser, or syntax analyzer, fits that stream of words to a grammatical model of the input language. The parser, in turn, invokes semantic elaboration routines to perform deeper analysis and to build structures that model the input program's actual meaning.

The scanner aggregates characters into words. For each word, it determines if the word is valid in the source language. For each valid word, it assigns the word a *syntactic category*, or part of speech.

Syntactic category
a classification of words according to their grammatical usage

In a practical sense, these categories correspond to terminal symbols in the language's grammar (see Section 3.2.2).

The scanner is the only pass in the compiler that manipulates every character of the input program. Because scanners perform a relatively simple task, grouping characters together to form words and punctuation in the source language, they lend themselves to fast implementations. Automatic tools for scanner generation are common. These tools process a mathematical description of the language's lexical syntax and produce a fast recognizer.

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00008-5

Nonetheless, many compilers use hand-crafted scanners because they are easily written and the resulting scanners can be fast and robust.

Conceptual Roadmap

To translate a program, the compiler must understand both its lexical structure—the spellings of the words in the program—and its syntactic structure—the grammatical way that words fit together to form statements and programs. This chapter describes the mathematical tools and programming techniques that are commonly used to construct scanners—programs that analyze the lexical structure of code.

Recognizer
a program that identifies specific words in a stream of characters

Scanner construction has a strong foundation in formal language theory. Scanners are based on *recognizers* that simulate deterministic finite automata. The compiler writer specifies the lexical structure of the input language using a set of regular expressions (REs). A series of constructions convert that specification into a scanner that reads a stream of characters and produces a stream of words, each tagged with its syntactic category from the programming language's grammar.

The technology to automate scanner construction has widespread application, from compilers to editors to filters for URLs in a web browser. Readily available tools use the well-developed theory of finite automata to build efficient recognizers from simple specifications.

Overview

Lexeme
the actual text for a word recognized by a scanner

A compiler's scanner reads the input stream and produces, as output, a stream of words, each labeled with its syntactic category—equivalent to a word's *part of speech* in English. Each time it is called, the scanner produces a pair, ⟨*lexeme*,*category*⟩, where *lexeme* is the spelling of the word and *category* is its syntactic category. This pair is sometimes called a *token*.

Microsyntax
the lexical structure of a language

To find and classify words, the scanner applies a set of rules that describe the lexical structure of the input programming language, sometimes called its *microsyntax*. Microsyntax specifies how to group characters into words and, conversely, how to separate words that run together. (In the context of scanning, punctuation marks and other symbols are treated as separate words.)

Western languages, such as English, have simple microsyntax. Adjacent alphabetic letters are grouped together, left to right, to form a word. A blank space terminates a word, as do most nonalphabetic symbols. (The word-building algorithm can treat a hyphen in the midst of a word as if it were an alphabetic character.) Once a group of characters has been aggregated together to form a potential word, the word-building algorithm can determine its validity and its potential parts of speech with a dictionary lookup.

Most programming languages have equally simple microsyntax. Characters are aggregated into words. In most languages, blanks and punctuation marks terminate a word. For example, ALGOL-60 and its descendants define an *identifier* as a single alphabetic character followed by zero or more alphanumeric characters. The identifier ends with the first nonalphanumeric character. Thus, `fee` and `f1e` are valid identifiers, but `12fum` is not. Notice that the set of valid words is specified by rules rather than by enumeration in a dictionary.

In a typical programming language, some words, called *keywords* or *reserved words*, match the rule for an identifier but have special meanings. Both `while` and `static` are keywords in both C and JAVA. Keywords (and punctuation marks) form their own syntactic categories. Even though `static` matches the rule for an identifier, the scanner in a C or JAVA compiler would undoubtedly classify it into a category that has only one element, the keyword `static`. The techniques used to implement a scanner must be capable of distinguishing individual words, such as keywords, without compromising the efficiency of recognizing the larger classes that contain them.

Keyword
a word that is reserved for a particular syntactic purpose and, thus, cannot be used as an identifier

The simple lexical structure of programming languages lends itself to efficient scanners. The compiler writer starts from a specification of the language's microsyntax. She either encodes the microsyntax into a notation accepted by a scanner generator, which then constructs an executable scanner, or she uses that specification to build a hand-crafted scanner. Both generated and hand-crafted scanners can be implemented to require just **O**(1) time per character, so they run in time proportional to the number of characters in the input stream.

This chapter begins, in Section 2.2, by introducing a model for *recognizers*, programs that identify words in a stream of characters. Section 2.3 describes a formal notation, *regular expressions*, for specifying both syntax and recognizers. Section 2.4 shows how to automate construction of an executable recognizer from an RE. Finally, Section 2.5 examines practical issues in scanner implementation, for both generated and hand-crafted scanners.

A Few Words About Time

Chapter 2 focuses on the design and implementation of a compiler's scanner. As such, it deals with issues that arise at three different times: design time, build time, and compile time.

At *design time*, the compiler writer creates specifications for the microsyntax of the programming language (the spelling of words). The compiler writer chooses an implementation method and writes any code that is needed.

At *build time*, the compiler writer invokes tools that build the actual executable scanner. For a specification-driven scanner, this process will include the use of tools to convert the specification into code and the use of a compiler to turn that code into an executable image.

At *compile time*, the end user invokes the compiler to translate application code into executable code. The compiler includes the scanner; it uses the scanner to convert the application code from a string of characters into a string of words, and to classify each of those words into a syntactic category or part of speech.

2.2 RECOGNIZING WORDS

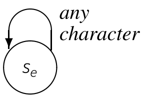
A character-by-character algorithm to recognize words is simple and understandable. The structure of the code can provide some insight into the underlying problem. Consider the problem of recognizing the keyword `new`, as shown in Fig. 2.1(a). The code assumes the presence of a routine *NextChar* that returns successive characters. The code tests for `n`, then `e`, then `w`. At each step, failure to match the appropriate character causes the code to reject the string and *try something else*. If the sole purpose of the program was to recognize the "`new`," then it should report an error. Because scanners rarely recognize only one word, we will leave this error path deliberately vague at this point.

We assume that, once *NextChar* reaches the end of the input file, it always returns `eof`.

The code fragment performs one test per character. We can represent the code fragment using the simple transition diagram shown in panel (b). The transition diagram represents the recognizer. Each labeled circle represents an abstract state in the computation. The initial state, or start state, is s_0. State s_3 is an accepting state; the recognizer reaches s_3 only when the input is "`new`." The double circle denotes s_3 as an accepting state.

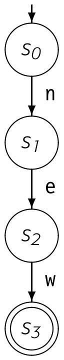
Arrows represent transitions from state to state based on input characters. If the recognizer starts in s_0 and reads the characters `n`, `e`, and `w`, the transitions take us to s_3. What happens on any other input, such as `n`, `o`, and `t`? The letter `n` takes the recognizer to s_1. The letter `o` does not match any edge leaving s_1. In the code, cases that do not match `new` execute *try something else*. The recognizer takes a transition to the error state. When we draw the transition diagram of a recognizer, we usually omit transitions to the error state. Each state has an implicit transition to the error state on each unspecified input.

Error state
s_e is a nonaccepting state with a transition back to itself on any character.



```
c ← NextChar();
if (c = 'n') then
    c ← NextChar();
    if (c = 'e') then
        c ← NextChar();
        if (c = 'w') then
            report success;
        else
            try something else;
    else
        try something else;
else
    try something else;
```

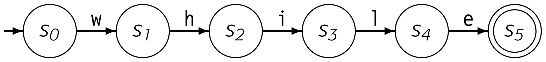
(a) Code



(b) Recognizer

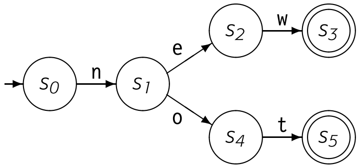
■ **FIGURE 2.1** Code Fragment to Recognize the Word "new".

Using this same approach to build a recognizer for `while` would produce the following transition diagram:



If the recognizer runs from s_0 to s_5, it has found the word `while`. The corresponding code would use five nested `if–then–else` constructs.

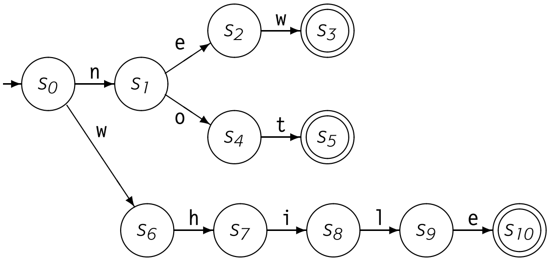
To recognize multiple words, the recognizer can have multiple edges that leave a given state. (In the code, these would become `else if` clauses.) The straightforward recognizer for both `new` and `not` is:



The recognizer uses a common test for `n` that takes it from s_0 to s_1, denoted $s_0 \xrightarrow{n} s_1$. If the next character is `e`, it takes the transition $s_1 \xrightarrow{e} s_2$. If, instead, the next character is `o`, it makes the move $s_1 \xrightarrow{o} s_4$. Finally, a `w` in s_2 causes the transition $s_2 \xrightarrow{w} s_3$, and a `t` in s_4 produces $s_4 \xrightarrow{t} s_5$. State s_3 indicates that the input was `new` while s_5 indicates that it was `not`. The recognizer takes one transition per input character.

Of course, the reader could renumber the recognizer's states without changing its "meaning." Renaming the states produces an equivalent recognizer.

We can combine the recognizer for `new` or `not` with the one for `while` by merging their initial states and relabeling all the states.



State s_0 has transitions for `n` and `w`. The recognizer has three accepting states, s_3, s_5, and s_{10}. If any state encounters an input character that does not match one of its transitions, the recognizer moves to the implicit error state, s_e.

The recognizer takes one transition for each input character. Assuming that we implement the recognizer efficiently, we should expect it to run in time proportional to the length of the input string.

2.2.1 A Formalism for Recognizers

Finite automaton
a formalism for recognizers that has a finite set of states, an alphabet, a transition function, a start state, and one or more accepting states

Transition diagrams serve as abstractions of the code that would be required to implement them. They can also be viewed as formal mathematical objects, called *finite automata*, that specify recognizers. Formally, a finite automaton (FA) is a five-tuple $(S, \Sigma, \delta, s_0, S_A)$, where

- S is the finite set of states in the recognizer, including s_e.
- Σ is the finite alphabet used by the recognizer. Typically, Σ is the union of the edge labels in the transition diagram.
- $\delta(s, c)$ is the recognizer's transition function. It maps each state $s \in S$ and each character $c \in \Sigma$ into some next state. In state s_i with input character c, the FA takes the transition $s_i \xrightarrow{c} \delta(s_i, c)$.
- $s_0 \in S$ is the designated start state.
- S_A is the set of accepting states, $S_A \subseteq S$. Each state in S_A appears as a double circle in the transition diagram.

Putting the FA for `new` or `not` or `while` into this formalism yields:

$$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$$

$$\Sigma = \{\texttt{e}, \texttt{h}, \texttt{i}, \texttt{l}, \texttt{n}, \texttt{o}, \texttt{t}, \texttt{w}\}$$

$$\delta = \left\{ \begin{array}{lllll} s_0 \xrightarrow{\texttt{n}} s_1, & s_0 \xrightarrow{\texttt{w}} s_6, & s_1 \xrightarrow{\texttt{e}} s_2, & s_1 \xrightarrow{\texttt{o}} s_4, & s_2 \xrightarrow{\texttt{w}} s_3, \\ s_4 \xrightarrow{\texttt{t}} s_5, & s_6 \xrightarrow{\texttt{h}} s_7, & s_7 \xrightarrow{\texttt{i}} s_8, & s_8 \xrightarrow{\texttt{l}} s_9, & s_9 \xrightarrow{\texttt{e}} s_{10} \end{array} \right\}$$

$$s_0 = s_0$$

$$S_A = \{s_3, s_5, s_{10}\}$$

δ is only partially specified. For all other combinations of state s_i and input character c, we define $\delta(s_i, c) = s_e$, where s_e is the error state. This quintuple is equivalent to the transition diagram; given one, we can easily recreate the other.

An FA accepts a string x if and only if, starting in s_0, the sequence of characters in x takes the FA through a series of transitions that leaves it in an accepting state when the entire string has been consumed. This corresponds to our intuition for the transition diagram. For the string `new`, our example recognizer runs through the transitions $s_0 \xrightarrow{\texttt{n}} s_1$, $s_1 \xrightarrow{\texttt{e}} s_2$, and $s_2 \xrightarrow{\texttt{w}} s_3$. Since $s_3 \in S_A$, and no input remains, the FA accepts `new`. For the input string `nut`, the behavior is different. On the letter `n`, the FA takes $s_0 \xrightarrow{\texttt{n}} s_1$. On `u`, it takes $s_1 \xrightarrow{\texttt{u}} s_e$. Once the FA enters s_e, it stays in s_e until it exhausts the input stream.

More formally, if the string x consists characters $x_1\, x_2\, x_3 \ldots x_n$ then the FA $(S, \Sigma, \delta, s_0, S_A)$ accepts x if and only if

$$\delta(\delta(\ldots\delta(\delta(\delta(s_0, x_1), x_2), x_3)\ldots, \mathrm{x}_{n-1}), x_n) \;\in\; S_A.$$

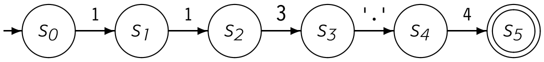
Intuitively, this definition corresponds to a repeated application of δ to a pair composed of some state s and input symbol x_i. The base case, $\delta(s_0, x_1)$, represents the FA's initial transition out of the start state, s_0, on the character x_1. The state $\delta(s_0, x_1)$ is then used as input to δ, along with x_2, which yields the next state, and so on, until all the input has been consumed. The result of the final application of δ is, again, a state. If that state is an accepting state, then the FA accepts $x_1\, x_2\, x_3 \ldots x_n$.

The FA can encounter a lexical error in the input. Some character x_j might take it into the error state s_e. Entry into s_e occurs because $x_1\, x_2\, x_3 \ldots x_j$ is not a valid prefix for any word in the language accepted by the FA. Alternatively, the FA can reach the end of its input while in a nonaccepting state. Either case indicates that the input string is not a word in the language.

Consider a string that causes an FA to halt in a nonaccepting state. If the FA passes through an accepting state on the way, then the string contains a prefix that is a valid word. As we will see in Section 2.4.5, scanners use this observation to find word boundaries.

2.2.2 Recognizing More Complex Words

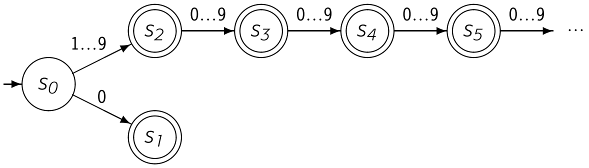
The character-by-character model shown in the original recognizer for `not` extends easily to handle arbitrary collections of fully specified words. How could we recognize a number with such a recognizer? A specific number, such as 113.4, is easy.



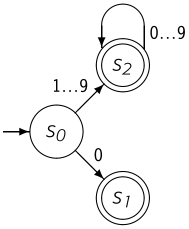
We will denote a range of characters with the first and last element, connected by an ellipsis, "...", as in 0...9.

Some systems and authors use a dash rather than an ellipsis. We use the ellipsis to avoid confusion with the minus sign.

To be useful, however, we need a transition diagram (and the corresponding code fragment) that can recognize any number. For simplicity's sake, we will limit the discussion to unsigned integers. In general, an integer is either zero, or it is a series of one or more digits where the first digit is from one to nine, and the subsequent digits are from zero to nine. (This definition rules out leading zeros.) How would we draw a transition diagram for this definition?



The transition $s_0 \xrightarrow{0} s_1$ handles the case for the digit zero. The other path, $\langle s_0, s_2, s_3, \ldots \rangle$, handles the case for a digit greater than zero. This path, however, violates the stipulation that S is finite.



Notice that all of the states on the path beginning with s_2 are equivalent, that is, they have the same labels on their output transitions and they are all accepting states. If we allow the transition diagram to have cycles, we can replace the path that starts at s_2 with a single transition from s_2 back to itself, as shown in the margin.

This cyclic transition diagram makes sense as an FA. From an implementation perspective, however, it is more complex than the acyclic transition diagrams shown earlier. We cannot translate this directly into a set of nested `if–then–else` constructs. The introduction of a cycle in the transition graph creates the need for cyclic control flow. We can implement this with a *while* loop, as shown in the code in Fig. 2.2(a). We can specify δ efficiently using a table:

δ	0	1	2	3	4	5	6	7	8	9	Other
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

```
state ← $s_0$;
char ← NextChar();

while (state $\neq s_e$ and char $\neq$ eof) do
    state ← $\delta$(state,char);
    char ← NextChar();
end;

if (state ∈ $S_A$) then
    report acceptance;
else report failure;
```

(a) Code to Interpret State Table

$$S = \{s_0, s_1, s_2, s_e\}$$

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

$$\delta = \left\{ \begin{array}{ll} s_0 \xrightarrow{0} s_1, & s_0 \xrightarrow{1\ldots 9} s_2 \\ s_2 \xrightarrow{0\ldots 9} s_2 & \end{array} \right\}$$

$$S_A = \{s_1, s_2\}$$

(b) Formal Definition of the FA

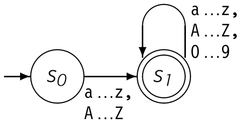
■ **FIGURE 2.2** A Recognizer for Unsigned Integers.

Changing the table allows the same basic code skeleton to implement other recognizers. Notice that this table has ample opportunity for compression. The columns for the digits *1* through *9* are identical, so they could be represented once, reducing the table to three columns: *0*, [*1* . . . *9*], and *other*. The code skeleton reports failure as soon as it enters s_e, so the table row for s_e is never used. If we elide the row for s_e, the table can be represented with three rows and three columns.

We can develop similar FAs for signed integers, real numbers, and complex numbers. A simplified version of the rule that governs identifier names in Algol-like languages, such as C or JAVA, might be:

> *an identifier consists of an alphabetic character followed by zero or more alphanumeric characters.*

This definition allows an infinite set of identifiers. It can be specified with the simple two-state FA shown in the margin. Many languages include designated special characters, such as _ and &, in the set of alphabetic characters.



FA for Identifiers

FAs can be viewed as specifications for a recognizer. However, they are not particularly concise specifications. To simplify scanner construction, we need a concise notation to specify the lexical structure of words, and a way to turn such a specification into an FA and into code to implement the FA. The remaining sections of this chapter develop precisely those ideas.

The FA for unsigned integers raises the distinction between a syntactic category, such as "unsigned integers," and a specific word, such as 113. The category is a set of one or more words; for example, both 12 and 113 might be members of the category "unsigned integer."

SECTION REVIEW

Recognizing and classifying words is a fundamental part of understanding the syntax of a program. For a given word, or a collection of words, the compiler writer can create a character-by-character recognizer. Such recognizers correspond to transition diagrams, which, in turn, correspond to finite automata.

Any finite set of words can be encoded in an acyclic transition diagram. Certain infinite sets of words, such as the set of integers or the set of identifiers in JAVA, can be encoded as well; they give rise to cyclic transition diagrams.

REVIEW QUESTIONS

1. Construct an FA for identifiers that consist of an alphabetic character followed by up to five alphanumeric characters.
2. Construct an FA for a PASCAL comment, which consists of an open brace, {, followed by zero or more characters drawn from the set Σ - }, followed by a close brace, }.

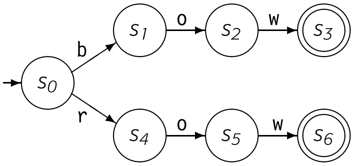
2.3 REGULAR EXPRESSIONS

The set of words accepted by a finite automaton, *F*, forms a language, denoted *L(F)*. The transition diagram of *F* specifies, in precise detail, how to spell every word in that language. Transition diagrams can be complex and nonintuitive. Thus, most systems use a notation called a *regular expression* (RE) to describe spelling. Any language described by an RE is considered a *regular language*.

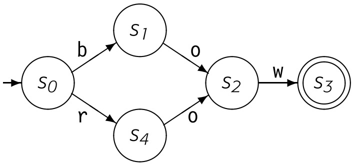
REs are equivalent to the FAs described in the previous section. (We will prove this with a construction in Section 2.4.) Simple recognizers have simple RE specifications.

- The language consisting of the single word `bow` can be described by an RE written as *bow*. Writing two characters next to each other implies that they are expected to appear in that order.
- The language consisting of the two words `bow` or `row` can be written as *bow* or *row*. To avoid possible misinterpretation of *or*, we write this using the symbol | to mean *or*. Thus, we write the RE as *bow* | *row*. We refer to | as an *alternation*.

For any given language, there may exist multiple REs that specify the language. For example, *bow* | *row* and (*b* |*r*) *ow* both specify the same language. The different REs, in turn, suggest different FAs, as shown below. The FAs shown in panels (a) and (b) accept the same language.



(a) FA suggested by *bow* | *row*



(b) FA suggested by (*b* | *r*) *ow*

Alternation is commutative, so *bow* | *row* describes the same language as *row* | *bow* or *(r* | *b) ow*.

To make this discussion concrete, consider some examples that occur in most programming languages. Punctuation marks, such as colons, semicolons, and various brackets, are represented by their character representations. Their REs have the same "spelling" as the punctuation marks themselves. Thus, the following REs might occur in the lexical specification for a programming language:

: ; ? => ( ) [ ] + //

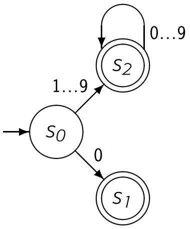
Similarly, keywords have simple REs.

if while this integer instanceof

To model syntactic categories that have large numbers of words, such as integers or identifiers, we need a way to denote an FA's cyclic edge.

The FA for an unsigned integer, shown in the margin, has three states: an initial state s_0, an accepting state s_1 for the unique integer zero, and another accepting state s_2 for all other integers. The key to this FA's power is the transition $s_2 \rightarrow s_2$ that occurs on each additional digit. State s_2 creates a rule to derive a new unsigned integer from an existing one: add another digit to the right end of the existing number. Another way of stating this rule is:

> *an unsigned integer is either a zero, or a nonzero digit followed by zero or more digits.*



FA for Unsigned Integers

To capture the essence of this FA, we need a notation for this notion of "zero or more occurrences" of an RE. For the RE *x*, we write this as *x**, with the meaning "zero or more occurrences of *x*." We call the * operator *Kleene*

Kleene closure
The RE *x** designates zero or more occurrences of *x*.

closure, or *closure* for short. Using the closure operator, we can write an RE for this FA:

$$0\,|\,(1\,|\,2\,|\,3\,|\,4\,|\,5\,|\,6\ \,|\,7\,|\,8\,|\,9)\ \ (0\,|\,1\,|\,2\,|\,3\,|\,4\,|\,5\,|\,6\,|\,7\,|\,8\,|\,9)^*.$$

Of course, we can write this RE more concisely as $0\,|\,(\,[1\ldots 9\,]\,)\,(\,[0\ldots 9\,]\,)^*$.

2.3.1 Formalizing the Notation

To work with REs in a rigorous way, we need a formal definition. Assume that we have an alphabet, Σ. An RE describes a set of strings over the characters in Σ, plus an additional character ϵ that represents the empty string. The set of strings defined by an RE is called a *language*. We denote the language described by some RE r as $L(r)$.

An RE is built up from three basic operations:

1. *Alternation* The alternation, or union, of two sets of REs, r and s, denoted $r\,|\,s$, is $\{x \mid x \in L(r) \text{ or } x \in L(s)\}$.
2. *Concatenation* The concatenation of two REs r and s, denoted rs, contains all strings formed by prepending a string from $L(r)$ onto one from $L(s)$, or $\{xy \mid x \in L(r) \text{ and } y \in L(s)\}$.
3. *Closure* The Kleene closure of r, denoted r^*, is $\bigcup_{i=0}^{\infty} r^i$. $L(r^*)$ contains all strings that consist of zero or more words from $L(r)$.

Finite closure
For any integer $i \geq 0$, the RE r^i designates one to i occurrences of r.

Positive closure
The RE r^+ denotes one or more occurrences of r.

For convenience, we sometimes use a *finite closure*. The notation r^i, $i \geq 0$, denotes from one to i occurrences of r. A finite closure can always be replaced with an enumeration of the possibilities; for example, r^3 is just $(\epsilon \mid r \mid rr \mid rrr)$ The *positive closure*, denoted r^+, is just rr^* and consists of one or more occurrences of r. Since the finite and positive closures can be rewritten in terms of alternation, concatenation, and Kleene closure, we ignore them in the discussion that follows.

Using alternation, concatenation, and Kleene closure, we can define the set of REs over an alphabet Σ as follows:

1. If $\mathtt{a} \in \Sigma$, then a is an RE denoting the set $\{\,\mathtt{a}\,\}$, and $L(a)$ is $\mathtt{a}$.
2. If r and s are REs, denoting sets $L(r)$ and $L(s)$, respectively, then
   $r \mid s$ is an RE denoting the alternation of $L(r)$ and $L(s)$;
   rs is an RE denoting the concatenation of $L(r)$ and $L(s)$; and
   r^* is an RE denoting the Kleene closure of $L(r)$.
3. ϵ is an RE denoting the set that only contains the empty string.

To eliminate ambiguities, parentheses have highest precedence, followed by closure, concatenation, and alternation, in that order.

2.3.2 Examples of Regular Expressions

The goal of this chapter is to show how we can use formal techniques to automate the construction of high-quality scanners and how we can encode the microsyntax of programming languages into that formalism. Before proceeding further, some examples from real programming languages are in order.

1. The rule given earlier for identifiers in Algol-like languages, an alphabetic character followed by zero or more alphanumeric characters, is just $([A \ldots Z] \mid [a \ldots z])\ ([A \ldots Z] \mid [a \ldots z] \mid [0 \ldots 9])^*$.

Most languages also allow a few special characters, such as _, %, $, or & in identifiers.

If the language limits the length of an identifier, we can use a finite closure, as in $([A \ldots Z] \mid [a \ldots z])\ ([A \ldots Z] \mid [a \ldots z] \mid [0 \ldots 9])^5$ for a six-character identifier.

2. An unsigned integer can be described as either zero or a nonzero digit followed by zero or more digits. The RE $0 \mid [1 \ldots 9]\,[0 \ldots 9]^*$ is more concise. The simpler specification $[0 \ldots 9]^+$ admits integers with leading zeroes such as 001.

3. Unsigned real numbers are more complex than integers. One possible RE might be $(0 \mid [1 \ldots 9]\,[0 \ldots 9]^*)\ (\epsilon \mid .\,[0 \ldots 9]^*)$. The first part is just the RE for an integer. The rest generates either the empty string or a decimal point followed by zero or more digits.

Programming languages often admit scientific notation, as in:

$$(0 \mid [1 \ldots 9]\,[0 \ldots 9]^*)\ (\epsilon \mid .\,[0 \ldots 9]^*)\ E\,(\epsilon \mid + \mid -)\ (0 \mid [1 \ldots 9]\,[0 \ldots 9]^*)$$

This RE describes a real number, followed by an E, followed by an integer to specify the exponent.

4. Quoted character strings have their own complexity. In most languages, a string can contain any character. While we can write an RE for strings using only the basic operators, it is our first example where a complement operator simplifies the RE. Using complement, a character string in C or JAVA can be described as " (^")* ".

C and C++ do not allow a string to span multiple lines—that is, if the scanner reaches the end of a line while inside a string, it terminates the string and issues an error message. If we represent newline with the escape sequence \n, in the C style, then the RE " (^(" | \n))* " will recognize a correctly formed C or C++ string and will take an error transition on a string that includes a newline.

Complement
The RE $^\wedge c$ specifies the set $\{\Sigma - c\}$, the complement of c with respect to Σ.

Complement has higher precedence than $*$, $|$, or $^+$.

Escape sequence
Two or more characters that the scanner translates into another character. Escape sequences are used for characters that lack a glyph, such as newline or tab, and for ones that occur in the syntax, such as an open or close quote.

REGULAR EXPRESSIONS IN VIRTUAL LIFE

Regular expressions are used in many applications to specify patterns in character strings. Some of the early work on translating REs into code was done to provide a flexible way of specifying strings in the "find" command of the QED text editor. From that early genesis, the notation has crept into many applications.

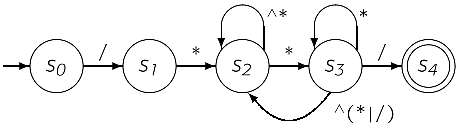
Unix and other operating systems use the asterisk as a wildcard to match substrings against file names. Here, ∗ is a shorthand for the RE Σ^*, specifying zero or more characters drawn from the entire alphabet of legal characters. (Since few keyboards have a Σ key, the shorthand has stayed with us.) Many systems use ? as a wildcard that matches a single character.

The `grep` family of tools, and their kin in non-Unix systems, implement regular expression pattern matching. (`grep` is an acronym for global regular-expression pattern match and print.)

Regular expressions have found widespread use because they are easily written and easily understood. They are one of the techniques of choice when a program must recognize a fixed vocabulary. They work well for languages that fit within their limited rules. They are easily translated into an executable form, and the resulting recognizer is fast.

5. Comments appear in a number of forms. C++ and JAVA offer two ways of writing a comment. The delimiter // indicates a comment that runs to the end of the current input line. The RE for this style of comment is straightforward: // (^\n)* \n.

Multiline comments in C, C++, and JAVA begin with the delimiter /∗ and end with ∗/. If we could disallow ∗ in a comment, the RE would be simple: /∗ (^∗)* ∗/. With ∗, the RE is more complex: /∗ (^∗ | ∗$^+$ ^/)* ∗/. An FA that implements this RE follows.

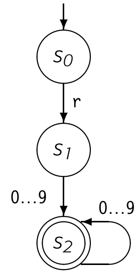


The relationship between the RE for multiline comments and its FA is less obvious than in many of the earlier examples.

The complexity of the RE and FA for multiline comments arises from the use of multicharacter delimiters. The transition from s_2 to s_3 encodes the fact that the recognizer has seen an ∗ so that it can handle either the appearance of a / or the lack thereof correctly.

By contrast, PASCAL used single-character comment delimiters: { and }, so a PASCAL comment is just { ^}* }.

In many cases, tighter specifications lead to more complex REs. Consider, for example, the register specifier in a typical assembly language. It consists of the letter `r` followed immediately by a small integer. In ILOC, which admits an unlimited set of register names, the RE might be $r[0\ldots 9]^+$, which corresponds to the FA shown in the margin. The FA accepts `r29` and rejects `s29`. It also accepts `r99999` even though no modern processor has 100,000 registers.

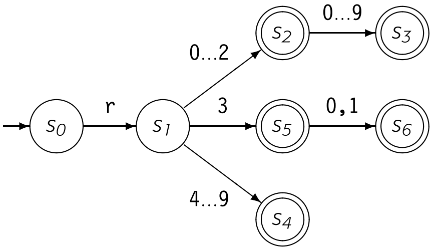


FA for Register Names

On a typical processor, the set of register names is severely limited—say, to 32, 64, 128, or 256 registers. With a more complex RE, the scanner can check the validity of a register name. For example, the RE

$$r\,(\,[0\ldots 2]\,(\,[0\ldots 9]\mid\epsilon\,)\mid[4\ldots 9]\mid(\,3\,(\,0\mid 1\mid\epsilon\,)))$$

specifies a much smaller language. It limits register numbers to the range [0,31] and allows an optional leading zero on single-digit register names. Thus, it accepts `r0`, `r00`, `r01`, and `r31`, but rejects `r001`, `r32`, and `r99999`. The corresponding FA looks like:



Which FA is better? They both make a single transition on each input character. Thus, they have the same cost, even though the second FA checks a more complex specification. The more complex FA has more states and transitions, so its representation requires more space. However, their operating costs are the same.

An alternative, of course, is to use the RE $r[0\ldots 9][0\ldots 9]$ and to test the register number as an integer.

This point is critical: the cost of operating an FA is proportional to the length of the input, not to the complexity of the RE or the number of states in the FA. More states need more space, but not more time. The build-time cost of *generating* the FA for a more complex RE may be larger, but the cost of operation remains one transition per character. A good implementation will have $O(1)$ cost per transition.

Can we improve the RE for a register name? The previous RE is both complex and counterintuitive. A simpler alternative might be:

r0 | r00 | r1 | r01 | r2 | r02 | r3 | r03 | r4 | r04 | r5 | r05 | r6 | r06 | r7 |
r07 | r8 | r08 | r9 | r09 | r10 | r11 | r12 | r13 | r14 | r15 | r16 | r17 | r18 |
r19 | r20 | r21 | r22 | r23 | r24 | r25 | r26 | r27 | r28 | r29 | r30 | r31

This RE is conceptually simpler, but much longer than the previous version. The resulting FA still requires one transition per input symbol. Thus, if we can control the growth in the number of states, we might prefer this version of the RE because it is clear and obvious. However, when processors suddenly have 256 or 384 registers, enumeration may become tedious, too.

2.3.3 Closure Properties of REs

Regular languages
Any language that can be specified by a regular expression is called a *regular language*.

REs and the languages that they generate have been the subject of extensive study. They have many interesting and useful properties. Some of these properties play a critical role in the constructions that build recognizers from REs.

REs are closed under many operations—that is, if we apply the operation to an RE or a collection of REs, the result is an RE. Obvious examples are concatenation, union, and closure. The concatenation of two REs x and y is just xy. Their union is $x \mid y$. The Kleene closure of x is just x^*. From the definition of an RE, all of these expressions are also REs.

These closure properties play a critical role in the use of REs to build scanners. Assume that we have an RE for each syntactic category in the source language, $a_0, a_1, a_2, \ldots, a_n$. Then, to construct an RE for all the valid words in the language, we can join them with alternation as $a_0 \mid a_1 \mid a_2 \mid \ldots \mid a_n$. Since REs are closed under union, the result is an RE. Anything that we can do to an RE for a single syntactic category will be equally applicable to the RE for all the valid words in the language.

URL-filtering software relies on this property to build fast FA-based recognizers.

Closure under union implies that any finite language is a regular language. We can construct an RE for any finite collection of words by listing them in a large alternation. Closure ensures that the alternation is an RE and that the corresponding language is regular.

Closure under concatenation lets us build complex REs from simpler ones by concatenating them. This property seems obvious. It lets us piece together REs in systematic ways. Closure ensures that ab is an RE as long as both a and b are REs. Thus, any technique that applies to either a or b applies to ab.

PROGRAMMING VERSUS NATURAL LANGUAGES

Lexical analysis highlights one of the subtle ways in which programming languages differ from natural languages, such as English or Portuguese. In natural languages, the relationship between a word's representation—its spelling or its pictogram—and its meaning is not obvious. In English, *are* is a verb while *art* is a noun, even though they differ only in the final character. Furthermore, not all combinations of characters are legitimate words. For example, *arz* differs minimally from *are* and *art*, but does not occur as a word in normal English usage.

A scanner for English could use FA-based techniques to recognize potential words, since all English words are drawn from a restricted alphabet. After that, however, it must look up the prospective word in a dictionary to determine if it is, in fact, a word. If the word has a unique part of speech, dictionary lookup will also resolve that issue. However, many English words can be classified with several parts of speech. Examples include *buoy* and *stress*; both can be either a noun or a verb. For these words, the part of speech depends on the surrounding context. In some cases, understanding the grammatical context suffices to classify the word. In other cases, it requires an understanding of meaning, for both the word and its context.

By contrast, the words in a programming language are almost always specified lexically. Thus, any string in $[1 \ldots 9][0 \ldots 9]^*$ is a positive integer. The RE $[a \ldots z]([a \ldots z] \mid [0 \ldots 9])^*$ defines a subset of the Algol identifiers; *arz*, *are* and *art* are all identifiers, with no lookup needed to establish the fact. To be sure, some identifiers may be reserved as keywords. However, these exceptions can be specified lexically, as well. No context is required.

This property results from a deliberate decision in programming language design. The choice to make spelling imply a unique part of speech simplifies scanning, simplifies parsing, and, apparently, gives up little in the expressiveness of the language. Some languages have allowed words with dual parts of speech—for example, PL/I has no reserved keywords. The fact that more recent languages abandoned the idea suggests that the complications outweighed any benefits of the extra flexibility.

REs are also closed under both Kleene closure and the finite closures. This property lets us specify particular kinds of large, or even infinite, sets with finite patterns. Kleene closure lets us specify infinite sets with concise finite patterns; examples include the integers and unbounded-length identifiers. Finite closures let us specify large but finite sets with equal ease.

The next section shows a sequence of constructions that build an FA from an RE. Section 2.6 shows an algorithm that builds an RE from an FA. Together, these algorithms establish the equivalence of REs and FAs. The fact that REs are closed under alternation, concatenation, and closure is critical to these constructions.

Complete FA
an FA that explicitly includes all error transitions

The equivalence between REs and FAs also suggests other closure properties. For example, given a complete FA, we can construct an FA that recognizes all words *w* that are not in *L*(FA), called the complement of *L*(FA). To build the FA for the complement, we can swap the designation of accepting and nonaccepting states in the original FA. Since FAs and REs are equivalent, this result shows that REs are closed under complement. Indeed, many systems that use REs include a complement operator, such as the ^ operator in `lex` and `flex`.

SECTION REVIEW

Regular expressions are a concise and powerful notation for specifying the microsyntax of programming languages. REs build on three basic operations over finite alphabets: alternation, concatenation, and Kleene closure. Other convenient operators, such as finite closures, positive closure, and complement, derive from the three basic operations. Regular expressions and finite automata are equivalent; any RE can be realized in an FA and the language accepted by any FA can be described with an RE. The next section formalizes that relationship.

REVIEW QUESTIONS

1. A six-character identifier might be specified with a finite closure:

   $$([A\ldots Z] \mid [a\ldots z])\,([A\ldots Z] \mid [a\ldots z] \mid [0\ldots 9])^5$$

   Rewrite the specification using only the three basic RE operations: alternation, concatenation, and Kleene closure.

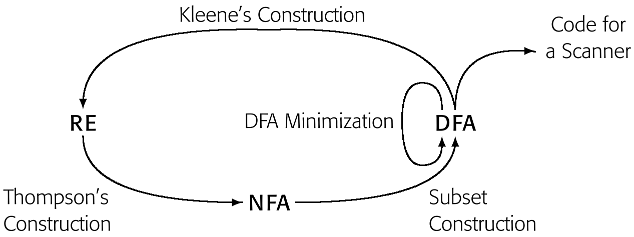
2. In PL/I, strings start and end with a quotation mark, ". In between are zero or more characters drawn from some alphabet, Σ. To represent ", the programmer writes two of them in a row. The string

   ```
   The quotation mark, ", should be typeset in italics.
   ```

   would be written in a PL/I program as

   ```
   "The quotation mark, "", should be typeset in italics."
   ```

   Design an RE and an FA to recognize PL/I strings.



■ **FIGURE 2.3** The Cycle of Constructions.

2.4 FROM REGULAR EXPRESSION TO SCANNER

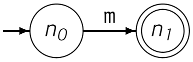
The goal of our work with finite automata is to automate the derivation of scanners from a set of REs. This section develops the constructions to transform an RE into an FA. The constructions rely on both *nondeterministic* FAs, or NFAs, and *deterministic* FAs, or DFAs. Kleene's construction, sketched in Section 2.6.1, builds an RE from any FA. Together, these constructions form a cycle, shown in Fig. 2.3.

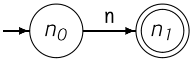
The cycle of constructions demonstrates that REs and FAs have equivalent expressive power. That is, an RE can express any language recognizable with an FA, and an FA can recognize any language that can be specified with an RE.

Section 2.4.1 explains the differences between an NFA and a DFA. Section 2.4.2 presents Thompson's construction, which builds an NFA directly from an RE. Section 2.4.3 presents the subset construction, which builds a DFA to simulate an NFA. Section 2.4.4 presents Hopcroft's algorithm for DFA minimization; an alternative minimization algorithm by Brzozowski appears in Section 2.6.3.

2.4.1 Nondeterministic Finite Automata

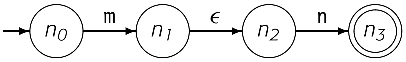
Recall from the definition of an RE that we designated the empty string, ϵ, as an RE. None of the FAs that we built by hand included ϵ, but some of the REs did. What role does ϵ play in an FA? We can use transitions on ϵ to combine FAs and to simplify construction of FAs from REs. For example, assume that we have FAs for m and n, called FA_m and FA_n.





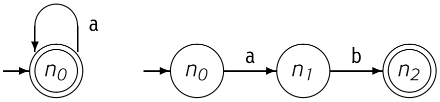
ϵ-transition
a transition on the empty string, ϵ, that does not advance the input

We can build an FA for *mn* by adding a transition on ϵ from the accepting state of FA_m to the initial state of FA_n, renumbering the states, and using FA_n's accepting state as the accepting state for the new FA.

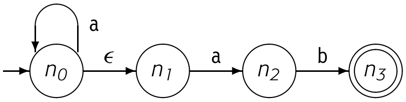


With an ϵ-transition, the definition of acceptance must change slightly to allow one or more ϵ-transitions between any two characters in the input string. For example, in state n_1, the FA takes the transition $n_1 \xrightarrow{\epsilon} n_2$ without consuming an input character. The change is minor and intuitive. It does, however, mean that an FA with ϵ-transitions can take multiple transitions per input character.

Merging two FAs with an ϵ-transition can complicate our model of how FAs work. Consider the FAs for the languages a^* and ab.



We can combine them with an ϵ-transition to form an FA for a^*ab.



The ϵ-transition, in effect, gives the FA two distinct transitions out of n_0 on the letter a. It can take the transition $n_0 \xrightarrow{a} n_0$, or it can take the two transitions $n_0 \xrightarrow{\epsilon} n_1$ and $n_1 \xrightarrow{a} n_2$. Which transition is correct? Consider the strings aab and ab. The FA should accept both strings. For aab, it should move: $n_0 \xrightarrow{a} n_0$, $n_0 \xrightarrow{\epsilon} n_1$, $n_1 \xrightarrow{a} n_2$, and $n_2 \xrightarrow{b} n_3$. For ab, it should move: $n_0 \xrightarrow{\epsilon} n_1$, $n_1 \xrightarrow{a} n_2$, and $n_2 \xrightarrow{b} n_3$.

Nondeterministic FA
an FA where the transition function can be multivalued and can include ϵ-transitions

Deterministic FA
an FA where the transition function is single-valued and does not include ϵ-transitions

As these two strings show, the correct transition out of n_0 on a depends on the characters that follow the a. At each step, an FA examines the current character. Its state encodes the left context, that is, the characters that it has already processed. Because the FA must make a transition before examining the next character, a state such as n_0 violates our notion of the behavior of a sequential algorithm. An FA that includes states such as n_0 that have multiple transitions on a single character is called a *nondeterministic finite automaton* (NFA). By contrast, an FA with unique character transitions in each state is called a *deterministic finite automaton* (DFA).

To make sense of an NFA, we need a set of rules that describe its behavior. Historically, two distinct models have been given for the behavior of an NFA.

1. Each time the NFA must make a nondeterministic choice, it follows the transition that leads to an accepting state for the input string, if such a transition exists. This model, using an omniscient NFA, is appealing because it maintains (on the surface) the well-defined accepting mechanism of the DFA. In essence, the NFA guesses the correct transition at each point.
2. Each time the NFA must make a nondeterministic choice, the NFA clones itself to pursue each possible transition. Thus, for a given input character, the NFA and its clones are in some set of states. In this model, the NFA pursues all paths concurrently.

   At any point, we call the specific set of states in which the NFA is active its *configuration*. When the NFA reaches a configuration in which it has exhausted the input and one or more of the clones has reached an accepting state, the NFA accepts the string.

Configuration of an NFA
the set of simultaneously active states of the NFA

In either model, the NFA $(S, \Sigma, \delta, n_0, S_A)$ accepts an input string $x = x_1\ x_2\ x_3 \ldots x_k$ if and only if there exists at least one path through the transition diagram that starts in n_0 and ends in some n_m such that the edge labels along the path match the input string, omitting ϵ's. In other words, the ith edge label must be x_i. This definition is consistent with either model of the NFA's behavior.

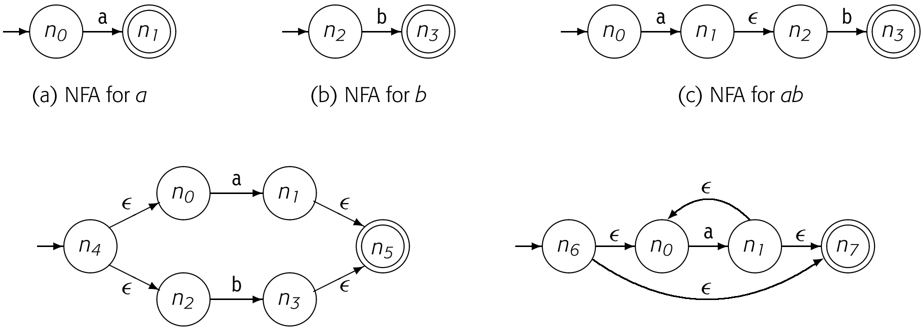
Equivalence of NFAs and DFAs

NFAs and DFAs are equivalent in their expressive power. Any DFA is a special case of an NFA. Thus, an NFA is at least as powerful as a DFA. Any NFA can be simulated by a DFA. The intuition behind this idea is simple; the construction, in Section 2.4.3, is a little more complex.

Consider the state of an NFA when it has reached some point in the input string. Under the second model of NFA behavior, the NFA has some finite set of operating clones. The set of states that those clones occupy form a *configuration* of the NFA. Each configuration is a subset of N, the set of states of the NFA. The number of such subsets is finite. Thus, an NFA with N states produces at most $|2^N|$ configurations.

Powerset of *N*
the set of all subsets of N, denoted 2^N

To simulate the behavior of the NFA, we need a DFA with a state for each valid configuration of the NFA. The resulting DFA may have exponentially more states than the NFA. Still, the number of configurations and, therefore, DFA states is finite. The DFA still makes one transition per input symbol, so it runs in time proportional to the length of the input string. Thus, the



(a) NFA for *a*

(b) NFA for *b*

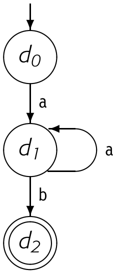
(c) NFA for *ab*

(d) NFA for *a* | *b*

(e) NFA for *a**

■ **FIGURE 2.4** Trivial NFAs for Regular Expression Operators.

simulation of an NFA on a DFA has a potential space problem, but not a time problem.



DFA for *aa***b*

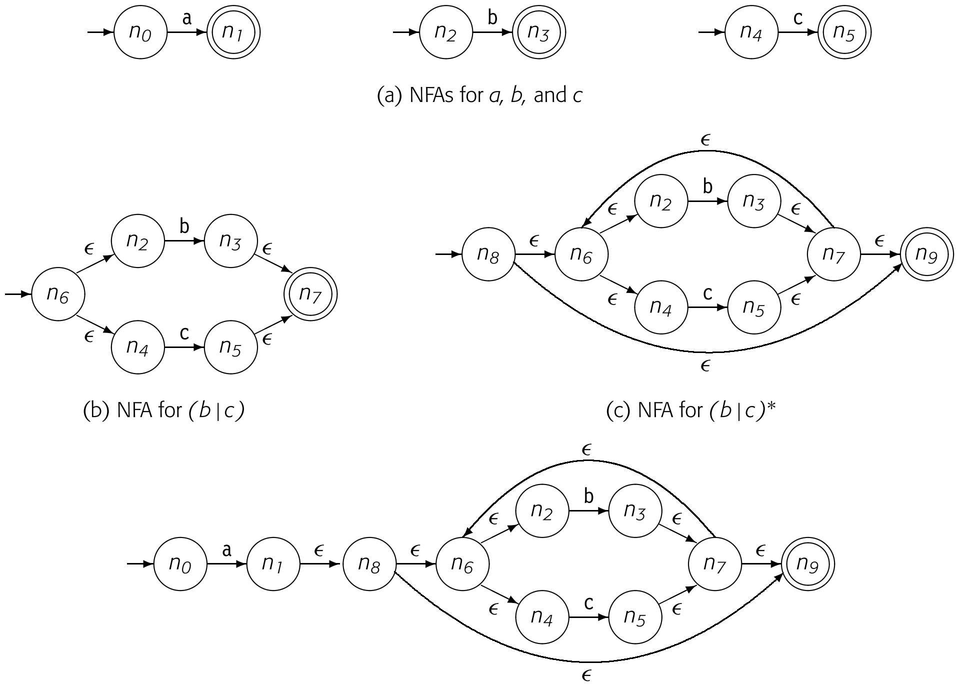
Since NFAs and DFAs are equivalent, we can construct a DFA that recognizes *a***ab*. The NFA that we saw earlier had two transitions out of n_0 on a. The obvious way to avoid this nondeterministic transition is to observe that *a***ab* is equivalent to *aa***b*. That RE suggests the DFA shown in the margin. The subset construction automates the transformation of an NFA into a DFA (see Section 2.4.3).

2.4.2 **RE to NFA: Thompson's Construction**

To put Thompson's construction in context, see the digression on page 718 in Section 14.2.

The first step in deriving a scanner from an RE constructs an NFA from the RE with *Thompson's construction*. The construction uses a simple, template-driven process to build up an NFA from smaller NFAs. It builds NFAs for individual symbols, $s \in \Sigma$, in the RE and applies transformations on the resulting NFAs to model the effects of concatenation, alternation, and closure. Fig. 2.4 shows the trivial NFAs for *a* and *b*, as well as the transformations to form *ab*, *a* | *b*, and *a** from NFAs for *a* and *b*. The transformations apply to arbitrary NFAs.

Fig. 2.5 shows the steps that Thompson's construction takes to build an NFA from the RE $a\,(b\,|\,c)^*$. First, the construction builds trivial NFAs for each character in the RE, shown in panel (a). It then applies the operators in precedence order. It builds an NFA for the parenthetic expression, (*b* | *c*), shown in panel (b). The closure is next, as shown in panel (c). Finally, it concatenates the NFA for *a* onto the front of the NFA for (*b* | *c*)*, shown in panel (d). This simple process produces an NFA for an RE written in terms of the three basic operators.



(a) NFAs for *a, b,* and *c*

(b) NFA for *(b | c)*

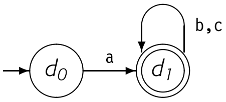
(c) NFA for *(b | c)**

(d) NFA for *a (b | c)**

■ **FIGURE 2.5** Applying Thompson's Construction to *a (b | c)**.

The NFAs derived from Thompson's construction have several specific properties that simplify an implementation. Each NFA has one start state and one accepting state. No transition, other than the initial transition, enters the start state. No transition leaves the accepting state. Finally, each state has at most two entering and two exiting ϵ-moves, and at most one entering and one exiting move on a symbol in the alphabet. Together, these properties simplify the representation and manipulation of the NFAs.

Fig. 2.5(d) shows the NFA that Thompson's construction builds for $a\ (b \mid c)^*$. The combination of the subset construction and DFA minimization should transform the NFA from Fig. 2.5(d) into a DFA similar to the one shown in the margin.



Minimal DFA for *a (b|c)**

2.4.3 **NFA to DFA: The Subset Construction**

Thompson's construction produces an NFA to recognize the language specified by an RE. Because DFA execution is much easier to simulate than NFA execution, the next step in building a recognizer from an RE converts the

```
$q_0 \leftarrow$ FollowEpsilon( $\{n_0\}$ )
$Q \leftarrow q_0$
WorkList $\leftarrow \{q_0\}$

while (WorkList $\neq \emptyset$) do
    remove q from WorkList

    for each character $c \in \Sigma$ do
        temp $\leftarrow$ FollowEpsilon(Delta(q, c))
        if temp $\neq \emptyset$ then
            if temp $\notin$ Q then
                add temp to both Q and WorkList
            T[q, c] $\leftarrow$ temp
```

■ **FIGURE 2.6** The Subset Construction.

NFA into an equivalent DFA. The algorithm to construct a DFA from an NFA is called the *subset construction*.

Valid configuration
configuration of an NFA that can be reached by some valid input string

The subset construction takes as input an NFA, $(N, \Sigma, \delta_N, n_0, N_A)$. It builds a model that captures all of the valid configurations that the NFA can enter in response to input strings. Each configuration contains one or more NFA states, all of which are reachable from the same collection of input strings. The model consists of two sets, Q and T. Each element q_i of Q is a set of NFA states that represents a valid configuration of the original NFA. T models the transitions between configurations that were discovered as the algorithm built Q.

To construct a DFA from the model, we create a DFA state d_i for each $q_i \in Q$. The DFA transitions follow directly from the transitions recorded in T. If the construction built q_j by considering how the NFA, in configuration q_i, would move on character c, then T contains the transition $q_i \xrightarrow{c} q_j$ and $\delta_N(d_i, c)$ should be d_j.

Fig. 2.6 shows the algorithm. It starts with a single configuration, q_0, constructed from n_0. It adds q_0 to Q and places it on *WorkList*. It uses *WorkList* to track which q_i must still be processed. The algorithm repeatedly removes a configuration, q_i, from the worklist. Then, for each $c \in \Sigma$, it computes the valid configuration, *temp*, reached by following transitions out of q_i on character c. If *temp* $\notin Q$, it adds *temp* to both Q and *WorkList*. It records the transition from q_i to *temp* in T. The algorithm halts when it exhausts the worklist.

Each set $q_i \in Q$ contains a set of core states and zero or more noncore states. The noncore states are reachable from a core state by following one or more ϵ-transitions. The model's initial configuration, q_0, has only one core state—

the NFA's initial state n_0. To form q_0, the algorithm adds all of the NFA states that n_0 implies—that is, those reachable along paths of ϵ-transitions. In the algorithm, the function *FollowEpsilon*(s) expands a set s with its noncore elements.

To compute the NFA configuration reachable from q_i on c, the algorithm applies δ_N to each $n_x \in q_i$. The function *Delta*(s,c) computes the core of a new configuration from s and c. If $\delta_N(n_x, c) = n_y$ and n_y is not the error state, then n_y is a core state in the new configuration. The algorithm uses *FollowEpsilon* to add the implied noncore elements.

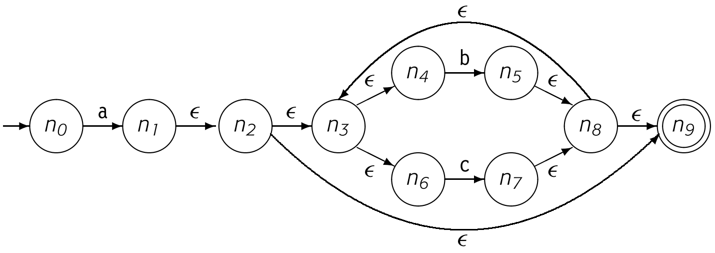
When the construction halts, Q contains all of the valid configurations of the NFA and T records all of the legal transitions between those configurations. Together, they represent a DFA that simulates the original NFA. To instantiate the DFA, we create a state d_i for each set $q_i \in Q$. If q_i contains an accepting state of the NFA—that is, some n_j such that $n_j \in N_A$—then the d_i that represents q_i is an accepting state—that is, $d_i \in D_A$. The DFA's transition function is built directly from T by mapping the sets in Q to their DFA states. Finally, q_0, the set constructed from n_0, becomes d_0, the DFA's initial state.

Notice that Q grows monotonically. The while loop adds sets to Q but never removes them. Since the number of configurations of the NFA is bounded—each q_i is a subset of 2^N (the powerset of N)—and each q_i appears on the worklist exactly once, the while loop must halt.

Q can become large—as large as $|2^N|$ distinct states. The amount of nondeterminism found in the NFA determines how much state expansion occurs. Recall, however, that the result is a DFA, so that it makes exactly one transition per input character, independent of the number of states in the DFA. Thus, any expansion introduced by the subset construction does not affect the asymptotic running time of the DFA. If the data structures used to represent the DFA become sufficiently large, memory locality may become an issue that affects runtime performance. Fortunately, DFA minimization and table compression can mitigate these effects (see Sections 2.4.4 and 2.5.4).

Example

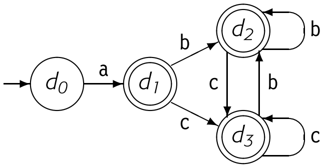
Fig. 2.7(a) shows the NFA that Thompson's construction built for $a\ (b \mid c)^*$ with its states renumbered to read left-to-right. The table in Fig. 2.7(b) sketches the steps of the subset construction on that NFA. The first column shows the name of the set in Q being processed in a given iteration of the while loop. The second column shows the name of the corresponding state in the new DFA. The third column shows the set of NFA states contained in the current set from Q. The final three columns show the result of applying *FollowEpsilon(Delta(*q_i, x*))* to the current set q_i and each character $x \in \Sigma$.



(a) Original NFA

			FollowEpsilon(Delta(q,x))		
Set Name	**DFA State**	**NFA States**	*a*	*b*	*c*
q_0	d_0	$\{ n_0 \}$	$\left\{\begin{matrix} n_1, n_2, n_3, \\ n_4, n_6, n_9 \end{matrix}\right\}$	– *none* –	– *none* –
q_1	d_1	$\left\{\begin{matrix} n_1, n_2, n_3, \\ n_4, n_6, n_9 \end{matrix}\right\}$	– *none* –	$\left\{\begin{matrix} n_5, n_8, n_9 \\ n_3, n_4, n_6 \end{matrix}\right\}$	$\left\{\begin{matrix} n_7, n_8, n_9 \\ n_3, n_4, n_6 \end{matrix}\right\}$
q_2	d_2	$\left\{\begin{matrix} n_5, n_8, n_9 \\ n_3, n_4, n_6 \end{matrix}\right\}$	– *none* –	q_2	q_3
q_3	d_3	$\left\{\begin{matrix} n_7, n_8, n_9 \\ n_3, n_4, n_6 \end{matrix}\right\}$	– *none* –	q_2	q_3

(b) Iterations of the Subset Construction



(c) Resulting DFA

■ **FIGURE 2.7** Applying the Subset Construction to the NFA from Fig. 2.5.

The algorithm takes the following steps:

1. The initialization sets q_0 to *FollowEpsilon*($\{n_0\}$), which is $\{\ n_0\ \}$.
2. The first iteration applies *Delta* and *FollowEpsilon* to q_0 with a, b, and c. Using a yields q_1, which contains six NFA states. Using b and c both produce the empty set.
3. The second iteration of the while loop examines q_1. Using a produces the empty set, while b yields q_2 and c yields q_3.
4. The third iteration of the while loop examines q_2. Using a produces the empty set, while b and c reconstruct q_2 and q_3.
5. The fourth iteration of the while loop examines q_3. It again reconstructs q_2 and q_3. The algorithm halts because *WorkList* is empty.

Fig. 2.7(c) shows the resulting DFA. The DFA states correspond to the sets in Q; the table and the transitions are taken from T. Each of q_1, q_2, and q_3 is an accepting state in the DFA because each contains n_9, the NFA's accepting state.

Fixed-Point Computations

The subset construction is an example of a *fixed-point computation*, a particular style of computation that arises regularly in many areas of computer science. These computations are characterized by the iterated application of a monotone function to some collection of sets drawn from a domain whose structure is known. These computations terminate when they reach a state where further iteration produces the same answer—a "fixed point" in the space of successive iterates. Fixed-point computations play an important and recurring role in compiler construction.

Monotone function
a function f on domain D is *monotone* if, $\forall\, x, y \in D, x \le y \Rightarrow f(x) \le f(y)$

Termination arguments for fixed-point algorithms usually depend on known properties of the domain. For the subset construction, the domain D is the set of subsets of N. The construction builds up the set Q, where each $q_i \in Q$ is a subset of N. Since N is finite, 2^N is finite and the number of distinct elements in Q is bounded.

The while loop functions as a monotone increasing function operating on Q. It adds elements to Q, so the successive iterations produce successively larger approximations to Q. If the ith approximation to Q is Q_i, then the algorithm ensures that $Q_i \le Q_{i+1}$. Because Q has, at most $|2^N|$ elements, the while loop can iterate at most $|2^N|$ times. It may, of course, reach a fixed point and halt more quickly than that.

A concern with fixed-point algorithms is the uniqueness of their results. For example, does the order in which the algorithm selects q from the worklist affect the final set Q? In this algorithm, the monotone function applies the union operator to sets. Because set union is both commutative ($a \cup b = b \cup a$) and associative ($(a \cup b) \cup c = a \cup (b \cup c)$), the order in which the loop adds sets to Q does not change the final result. The subscripts assigned to specific $q_i \in Q$ may change with different orders of removal from the worklist, but the final Q will always contain the same sets of valid NFA configurations. The different possibilities for Q differ, at most, by the names of the q_i sets.

2.4.4 DFA to Minimal DFA

As the final step in the RE→DFA construction, we can employ an algorithm to minimize the number of states in the automaton. The subset construction can produce a DFA that has a large set of states. While the size of the DFA does not affect its asymptotic complexity, it does determine the recognizer's footprint in memory. On modern computers, the speed of memory accesses often governs the speed of computation. A smaller recognizer may fit better into the processor's lowest level of cache memory, producing faster average accesses.

To reduce the number of states in the DFA, the scanner generator can apply a DFA minimization algorithm. The best known and asymptotically fastest algorithm, Hopcroft's algorithm, constructs a minimal DFA from an arbitrary DFA by grouping together states into sets that are *equivalent*. Two DFA states are equivalent when they produce the same behavior on any input string. The algorithm finds the largest possible sets of equivalent states; each set becomes a state in the minimal DFA.

Set partition
A *partition* of S is a collection of disjoint, nonempty subsets of S whose union is exactly S.

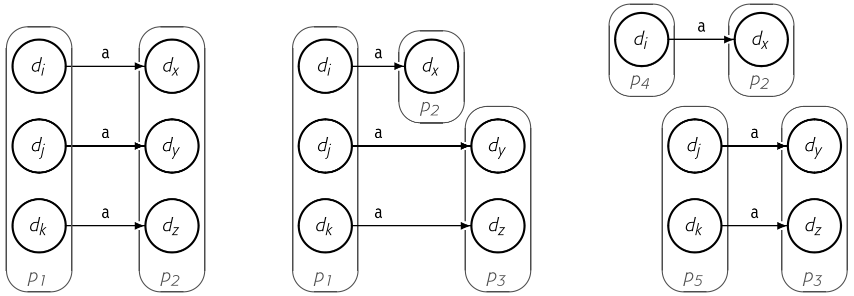
The algorithm constructs a *set partition*, $P = \{p_1, p_2, p_3, \ldots, p_m\}$ of the DFA states. Each p_i contains a set of equivalent DFA states. More formally, it constructs a partition with the smallest number of sets, subject to the following two rules:

1. $\forall\, c \in \Sigma$, if $d_i, d_j \in p_s$; $d_i \xrightarrow{c} d_x$; $d_j \xrightarrow{c} d_y$; and $d_x \in p_t$; then $d_y \in p_t$.
2. If $d_i, d_j \in p_k$ and $d_i \in D_A$, then $d_j \in D_A$.

Rule 1 mandates that two states in the same set must, for every character $c \in \Sigma$, transition to states that are, themselves, members of a single set in the partition. Rule 2 states that any single set contains either accepting states or nonaccepting states, but not both.

P_0 divides D into accepting and nonaccepting states, a fundamental difference in behavior specified by rule 2.

These two properties not only constrain the final partition, P, but they also lead to a construction for P. The algorithm starts with the coarsest partition on behavior, $P_0 = \{\, D_A, \{\, D - D_A \,\}\, \}$. It then iteratively "refines" the partition until both properties hold true for each set in P. To refine the partition, the algorithm splits sets based on the transitions out of DFA states in the set.



(a) a Does Not Split p_1 (b) a Splits p_1 (c) Sets from Panel b After the Split

■ **FIGURE 2.8** Splitting a Set Around a.

Fig. 2.8 shows how the algorithm uses transitions to split sets in the partition. In panel (a), all three DFA states in set p_1 have transitions to DFA states in p_2 on the input character a. Specifically, $d_i \xrightarrow{\texttt{a}} d_x$, $d_j \xrightarrow{\texttt{a}} d_y$, and $d_k \xrightarrow{\texttt{a}} d_z$. Since $d_i, d_j, d_k \in p_1$, and $d_x, d_y, d_z \in p_2$, sets p_1 and p_2 conform to rule 1. Thus, the states in p_1 are behaviorally equivalent on a, so a does not induce the algorithm to split p_1.

By contrast, panel (b) shows a situation where the character a induces a split in set p_1. As before, $d_i \xrightarrow{\texttt{a}} d_x$, $d_j \xrightarrow{\texttt{a}} d_y$, and $d_k \xrightarrow{\texttt{a}} d_z$, but $d_x \in p_2$ while $d_y, d_z \in p_3$. This situation violates rule 1, so a induces the algorithm to split p_1 into two sets, $p_4 = \{d_i\}$ and $p_5 = \{d_j, d_k\}$, shown in panel (c).

The algorithm, shown in Fig. 2.9, builds on these ideas. Given a DFA, it constructs an initial partition of the DFA's states. It starts with the coarsest partition consistent with rule 2, $\{\, D_A, \{\, D - D_A \,\}\}$.

Starting with the largest possible sets and splitting them is an *optimistic* approach to building the sets. Optimism is discussed in Section 9.3.6 or [359].

This choice of the initial partition has two consequences. First, since each set in the final partition is constructed by splitting a set in an earlier approximation, it ensures that no set in the final partition will contain both accepting and nonaccepting states. Second, by choosing the largest sets consistent with rule 2, it imposes the minimum constraints on the splitting process which, in turn, can lead to larger sets in the final partition. (Larger sets means fewer states in the final DFA.)

The algorithm operates from a worklist of states, starting with the initial partition $\{\, D_A, \{\, D - D_A \,\}\}$. It repeatedly picks a set s from the worklist and uses that set to refine *Partition* by splitting sets based on their transitions into s.

To identify states that must split because of a transition into set s on some character c, the algorithm inverts the transition function. It computes *Image*

```
Partition $\leftarrow \{D_A,\ \{D - D_A\}\}$
Worklist $\leftarrow \{D_A,\ \{D - D_A\}\}$

while ( Worklist $\neq \emptyset$ ) do
    select a set s from Worklist and remove it
    for each character $c \in \Sigma$ do
        Image $\leftarrow \{x \mid \delta(x,c) \in s\}$
        for each set $q \in$ Partition that has a state in Image do
            $q_1 \leftarrow q \cap$ Image
            $q_2 \leftarrow q - q_1$
            if $q_2 \neq \emptyset$ then                // split q around s and c
                remove q from Partition
                Partition $\leftarrow$ Partition $\cup\ q_1 \cup q_2$
                if $q \in$ Worklist then         // and update the Worklist
                    remove q from Worklist
                    WorkList $\leftarrow$ WorkList $\cup\ q_1 \cup q_2$
                else if $|q_1| \leq |q_2|$ then
                    WorkList $\leftarrow$ Worklist $\cup\ q_1$
                else WorkList $\leftarrow$ WorkList $\cup\ q_2$
                if $s = q$ then                  // need another s
                    break
```

■ **FIGURE 2.9** DFA Minimization Algorithm.

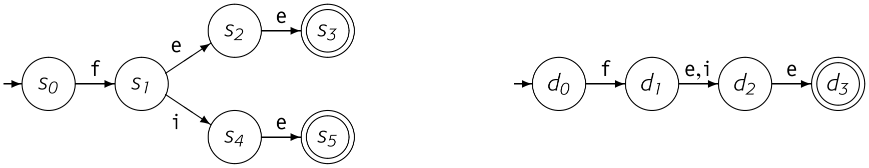
as the set of DFA states that can reach a state in s on a transition labeled c. It then systematically examines each set q that has a state in *Image* to see if *Image* induces a split in q. If *Image* divides q into nonempty sets q_1 and q_2, it replaces q in *Partition* with q_1 and q_2.

All that remains, in processing q with respect to c, is to update the worklist. If q is on the worklist, then the algorithm replaces q with both q_1 and q_2. The rationale is simple: q was on the worklist for some potential effect; that effect might be from some character other than c, so all of the DFA states in q need to be represented on the worklist.

If, on the other hand, q is not on the worklist, then the only effect that splitting q can have on other sets is to split them. Assume that some set r has transitions on letter e into q. Dividing q might create the need to split r into sets that transition to q_1 and q_2. In this case, either of q_1 or q_2 will induce the split, so the algorithm can choose between them. Using the smaller set will lead to faster execution; for example, computing *Image* takes time proportional to the size of the set.

Step	Partition on Entry	Worklist	s	c	Image	q	q_1	q_2	Action
–	p_0: $\{s_3, s_5\}$, p_1: $\{s_0, s_1, s_2, s_4\}$	p_0, p_1	—	—	—	—	—	—	—
1	p_0: $\{s_3, s_5\}$, p_1: $\{s_0, s_1, s_2, s_4\}$	p_1	p_0	e	s_2, s_4	p_1	s_2, s_4	s_0, s_1	*split* $p_1 \rightarrow p_2, p_3$
		p_2, p_3	p_0	f	∅	∅	∅	∅	*none*
		p_2, p_3	p_0	i	∅	∅	∅	∅	*none*
2	p_0: $\{s_3, s_5\}$, p_2: $\{s_2, s_4\}$, p_3: $\{s_0, s_1\}$	p_3	p_2	e	s_1	p_3	s_1	s_0	*split* $p_3 \rightarrow p_4, p_5$
		p_4, p_5	p_2	f	∅	∅	∅	∅	*none*
		p_4, p_5	p_2	i	s_1	∅	∅	∅	*none*
3	p_0: $\{s_3, s_5\}$, p_2: $\{s_2, s_4\}$, p_4: $\{s_1\}$, p_5: $\{s_0\}$	p_5	p_4	e	∅	∅	∅	∅	*none*
		p_5	p_4	f	s_0	p_5	s_0	∅	*none*
		p_5	p_4	i	∅	∅	∅	∅	*none*
4	p_0: $\{s_3, s_5\}$, p_2: $\{s_2, s_4\}$, p_4: $\{s_1\}$, p_5: $\{s_0\}$	∅	p_5	e	∅	∅	∅	∅	*none*
		∅	p_5	f	∅	∅	∅	∅	*none*
		∅	p_5	i	∅	∅	∅	∅	*none*

(a) Iterations of Hopcroft's Algorithm on the Original DFA for "fee | fie"



(b) Original DFA for "fee | fie"

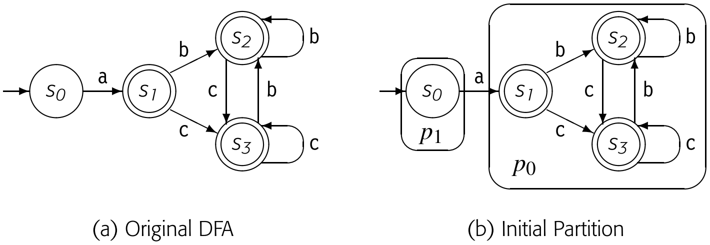
(c) Minimal DFA for "fee | fie"

■ **FIGURE 2.10** Applying the DFA Minimization Algorithm.

To construct the new DFA from the final *Partition*, we can create a state to represent each set $p_i \in$ *Partition* and add transitions between the new representative states as needed. For the state for p_m, we add a transition to the state for p_n on character c if some $d_j \in p_m$ has a transition to some $d_k \in p_n$ on c. The construction ensures that, if $d_j \xrightarrow{c} d_k$, where $d_j \in p_m$ and $d_k \in p_n$, then every state in p_m has a similar transition on c to a state in p_n. If this condition did not hold, the algorithm would have split p_m around the transitions on c. The resulting DFA is minimal; the proof is beyond our scope.

Examples

As a first example, consider the DFA for *fee* | *fie* shown in Fig. 2.10(b). Panel (a) shows the progress of Hopcroft's algorithm on this DFA.



■ **FIGURE 2.11** DFA for $a\,(b \mid c)^*$.

The first line shows the algorithm's initial configuration. *Partition* and *Worklist* each contain two sets: $\{\, D_A, \{D - D_A\}\,\}$. D_A is labeled p_0 while $\{D - D_A\}$ is labeled p_1.

The algorithm enters the while loop and removes p_0 from *Worklist*; it becomes s. The algorithm iterates over the characters in Σ, in the order e, f, and i. For e, p_0 splits p_1 into two sets: p_2: $\{s_0, s_1\}$ and p_3: $\{s_2, s_4\}$. The algorithm removes p_1 from *Partition* and adds p_2 and p_3. Next, it removes p_1 from *Worklist* and adds p_2 and p_3. For f and i, no edges enter p_0. Thus, *Image* is empty and no splits occur.

The second iteration proceeds in a similar fashion. It chooses p_2. The character e splits p_3 and causes an update to both *Partition* and *Worklist*. For f, the *Image* set is empty. For i, the *Image* set contains s_1. Because the algorithm already split p_3 around e, this situation does not cause a split. It will, however, add a transition to the final DFA.

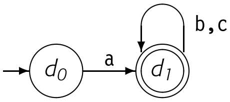
The third iteration chooses p_4 from the worklist. Both e and i produce empty *Image* sets. With f, the *Image* set contains s_0. Because p_5, which contains s_0, is a singleton set, it cannot be split. This situation, however, will add a transition to the final DFA.

The final iteration takes p_5 from the worklist. For each of e, f, and i, the *Image* set is empty. Thus, the iteration splits no sets, adds no transitions, and empties the worklist. Panel (c) shows the minimal DFA.

As a second example, consider the DFA for $a\,(b \mid c)^*$ produced by Thompson's construction and the subset construction, shown in Fig. 2.11(a). The initial partition is $\{\, p_0 : \{s_1, s_2, s_3\}, p_1 : \{s_0\}\,\}$.

The algorithm first selects p_0 and examines each of a, b, and c. For a, *Image* contains s_0 which is in a singleton set, p_1. Thus, a introduces a transition for the final DFA, but no split. For both b and c, *Image* is $\{s_1, s_2, s_3\}$, which is exactly p_0. Thus, q_2 is empty and no splits occur.

Next the algorithm removes p_1 and examines each of a, b, and c. Since no transitions enter p_1, the *Image* set is empty for each letter. No further splits occur. The original two set partition is the final partition. The final DFA has two states, as shown in the margin. Recall that this is the DFA that we suggested a human would derive. After minimization, the automatic techniques produce the same result.



Minimal DFA for *a* (*b*|*c*)*

2.4.5 Using a DFA as a Scanner

The tools in the three previous sections provide an algorithmic path from an RE to a minimal DFA. As we saw in Fig. 2.2, a DFA can be simulated with a simple table-driven skeleton. Taken together, these suggest that we can automate scanner construction by taking REs for all of the words in a programming language, combining them into a single RE, and using the resulting DFA to build a scanner. Reality, however, is more complex. Scanners and DFAs differ in two critical ways that affect how we formulate and build RE-based scanners.

Model of Execution

A DFA reads all of its input and accepts the input if its last state is a final state. That is, a DFA tries to find one word. By contrast, a scanner reads enough input to find the next word in the input stream. The scanner leaves the input stream in a state from which it can find the next word.

This difference necessitates a new model of execution. Rather than exhausting the input stream, the scanner simulates the DFA until it hits an error—that is, until it is in some state d_i with input character c such that $\delta(d_i, c) = s_e$, the error state. We also define $\delta(d_j, \texttt{eof}) = s_e, \forall\, d_j \in D$.

If d_i is an accepting state, $d_i \in D_A$, the scanner has found a word. If d_i is not an accepting state, the scanner may have passed through such a state on its way to d_i. To determine if it did, the scanner must back up, one character at a time until it either reaches an accepting state or it exhausts the lexeme.

This scheme adds some work to the implementation. The scanner must either record states or invert δ. Either approach takes time proportional to the number of scanned characters. A character may be scanned multiple times; Section 2.5.1 shows a method for avoiding the worst case of this behavior.

Finding Syntactic Categories

A DFA returns a binary answer: it either accepts or rejects the input. By contrast, a scanner returns a token, ⟨*lexeme*,*category*⟩, that gives the spelling

IDENTIFYING KEYWORDS

Most programming languages reserve the keywords that identify critical parts of the syntax, words such as `if`, `then`, and `while`. In a typical scanner and parser, each keyword has a unique syntactic category with just one lexeme. The compiler writer faces a choice: specify each keyword with its own rule, or fold keywords into the rule for identifiers and recognize them with some other mechanism. Either approach works and can lead to an efficient scanner.

With a separate rule for each keyword, the scanner can return the appropriate category using the same mechanism used for other categories, such as *number* or *identifier*. The extra rules may add minor cost to scanner generation and the DFA may have more states. However, since the process produces a DFA, the resulting scanner will still require $\mathbf{O}(1)$ time per character.

As an alternative, most scanners build a table of all identifier names. This table serves as a start on the compiler's *symbol table* and as a way to map identifier names into small integers so that they can be represented and compared efficiently (see Section 4.5). If the compiler writer preloads the symbol table with the keywords and their syntactic categories, the scanner will find the keywords as previously seen and categorized identifiers, and will return the appropriate category for each.

and syntactic category of the next word. If the scanner encounters an error, it can return an invalid token.

$d_i \in D_A$ maps uniquely to a category, but one category may map to multiple d_is.

If we construct the DFA so that each final state maps to a single category, then the scanner can find the category with a simple table lookup. However, this scheme requires that we build the DFA in a way that preserves the mapping of final states to categories.

Most scanner generators take, as input, a list of REs, $r_1, r_2, \ldots, r_k$, each of which defines the spelling of some category. The obvious way to build a single DFA is to construct a single RE, $(r_1 \mid r_2 \mid \ldots \mid r_k)$, and construct a DFA from this RE. However, Thompson's construction will immediately unify the final states, destroying the mapping from $d_i \in D_A$ to categories.

To preserve the mapping from final states to unique categories, the scanner generator can build a distinct NFA for each rule, using Thompson's construction. It can join those NFAs into a single DFA, with a new start state and ϵ-transitions, and use the subset construction to build a DFA that simulates the NFA. The resulting DFA may have more states, but each final state corresponds to a single rule and, therefore, one syntactic category.

If two rules overlap, the subset construction will merge their final states. When the subset construction merges final states, the scanner generator must decide which syntactic category it will return for that final state. In practice, scanner generators let the compiler writer specify a precedence among syntactic categories. The scanner generator assigns to the final state the category with the highest precedence.

This situation reveals an ambiguity in the specification. For example, a keyword such as `then` may also match the rule for an identifier.

Both `flex` and `lex` assign higher precedence to the rule that appears first in the list of rules.

Minimization poses another challenge. Hopcroft's algorithm immediately combines all of the final states into a single partition, destroying the property that final states map to syntactic categories. If, however, the scanner generator constructs an initial partition that places the final states for each syntactic category in a distinct set in the initial partition, then the rest of the algorithm will maintain that property.

Hopcroft's algorithm splits partitions but never combines them.

The resulting DFA may be larger than the minimal DFA that results from grouping all final states into the same partition. However, the larger DFA has the property that the compiler needs: each final state maps to a specific syntactic category.

The Role of Whitespace

Programmers often refer to blanks and tabs, when used to format code, as *whitespace*. In most languages, whitespace has no intrinsic meaning. Scanners for these languages typically recognize and discard whitespace. The primary impact of whitespace arises from its absence in the REs that define words in the language.

For example, the fact that the RE for an identifier or keyword name does not include a blank or tab forces an RE-based scanner to separate `do` and `i` in a sentence such as:

```
do i = 1 to 100
```

For similar reasons, the RE for an identifier does not contain `+`, `-`, `*`, or `/`. This fact ensures that "`a * b`" scans the same as "`a*b`".

FORTRAN 66

In FORTRAN 66, blanks are not significant. That is, "`n a m e`" and "`name`" refer to the same identifier. This rule complicates scanning. The header of a FORTRAN do loop might read:

```
do 10 i = 1,100
```

where `10` is the label of the last statement in the loop body, `i` is the loop's index variable, and `i`'s value runs from `1` to `100`. (The increment defaults to one unless specified.)

Of course, `do10i` is a valid variable name. To differentiate between these two statements:

```
do 10 i = 1
do 10 i = 1,100
```

a scanner must read beyond the `=` and `1` to the comma. The comma proves that the second statement is a loop header, and the scanner can separate `do10i` into three words, `do`, `10`, and `i`. Few, if any languages, have followed FORTRAN's example.

PYTHON

PYTHON takes the opposite approach: not only are blanks significant, but the number of blanks at the start of a line determines the meaning of a PYTHON program. Rather than using bracket constructs, such as { and } or `begin` and `end`, to indicate block structure, PYTHON relies on changes in indentation.

A simple way of handling leading blanks in PYTHON is to add a rule that recognizes an end-of-line followed by zero or more blanks. The scanner can then test the length of the lexeme. If its length is identical to the previous token in this category, it returns the result of calling the scanner again. If its length differs, the scanner can return a category indicating the start of a block or the end of a block, as appropriate.

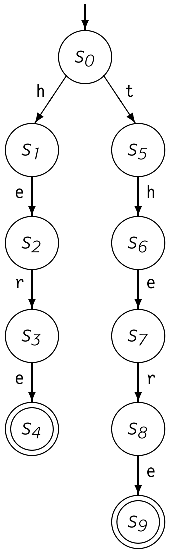
SECTION REVIEW

Given a regular expression, we can derive a DFA to recognize the language specified by the RE in a two-step process: (1) apply Thompson's construction to build an NFA for the RE and (2) use the subset construction to construct a DFA that simulates the NFA. The resulting DFA will be both fast and efficient.

To build a scanner that recognizes multiple categories of words, each specified by an RE, we can use the two-step process to build a minimal RE for each category, and then combine those DFAs into a single NFA by adding a new start state that has ϵ-transitions to the start states of the individual minimal DFAs. The subset construction will convert that NFA to a DFA that has distinct final states for each category.

REVIEW QUESTIONS

1. Consider the RE *who* | *what* | *where*. Use Thompson's construction to build an NFA from the RE. Use the subset construction to build a DFA from the NFA. Minimize the DFA.
2. Minimize the DFA shown in the margin.



DFA for Review Question 2

2.5 IMPLEMENTING SCANNERS

Scanner generators apply the theory of formal languages directly to the problem of creating efficient tokenizers for programming languages. Using the techniques outlined in the previous section, these tools build DFA models of a programming language's microsyntax and convert those models into executable code.

This section discusses three implementation strategies for converting a DFA into executable code. The first two, table-driven scanners and direct-coded scanners, rely on *scanner generators* to translate a set of REs into a scanner. The third strategy has the compiler writer craft a custom scanner by hand. Each approach can lead to a robust and efficient scanner.

Generated scanners operate by simulating a DFA, as described in Section 2.4.5. They begin in the initial state and take a series of transitions based on the current state and input character. When the current state has no valid transition for the input character, the scanner backs up until it finds an accepting state. If it cannot find an accepting state, it reports a lexical error.

The next three subsections discuss implementation differences between table-driven, direct-coded, and hand-coded scanners. The strategies differ in how they model the DFA's transition structure and how they simulate its operation. Each approach can produce a scanner that uses $O(1)$ time per character; the differences, however, can affect the constants in the complexity equation. The final subsection looks at two different approaches for handling reserved keywords.

2.5.1 Table-Driven Scanners

The table-driven approach uses a skeleton scanner for control and a set of generated tables to encode language-specific knowledge. Typically, table-driven scanners come from scanner generators. At design time, the compiler writer creates a set of REs. At build time, the scanner generator creates a set of tables to implement the DFA and compiles the tables with the skeleton scanner.

```
state ← $s_0$;
lexeme ← "";
clear stack;
push(bad);

while (state ≠ $s_e$) do
    char ← NextChar();
    lexeme ← lexeme + char;

    if state ∈ $S_A$ then
        clear stack;
        push(bad);

    push(state);

    col ← CharClass[char];
    state ← $\delta$[state,col];

while (state ∉ $S_A$ and state ≠ bad) do
    state ← pop();
    if state ≠ bad then
        truncate lexeme;
        RollBack();

if state ∈ $S_A$
    then return Type[state];
    else return invalid;
```

(a) Code to Interpret the Tables

r	0...9	Other
Register	*Digit*	*Other*

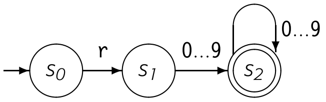
(b) The Classification Table, *CharClass*

	Register	Digit	Other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

(c) The Transition Table, δ

s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

(d) The Token Type Table, *Type*



(e) The Underlying DFA

■ **FIGURE 2.12** A Table-Driven Scanner for Register Names.

Conceptually, the process looks like:

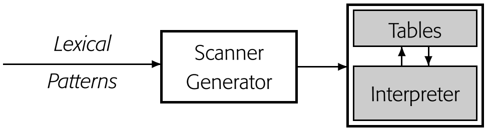


Fig. 2.12 shows a table-driven scanner for the RE $r\,[0\ldots9]^{+}$, introduced in Section 2.3.2. Panel (a) shows the code for the skeleton scanner. Panels (b), (c), and (d) show the tables that encode the DFA, which is shown in panel (e). The code is more detailed than that shown in Fig. 2.2 on page 35, but the basic operation is similar.

In the table, the entries for 0...9 have been combined into a single column.

Section 2.5.4 discusses table compression.

The skeleton scanner divides into four sections: initializations, a scanning loop to model the DFA's behavior, a roll back loop to find a final state, and a final section to interpret and report the results. The scanning loop repeats the two basic actions of a scanner: read a character and take a transition. The loop halts when the DFA enters the error state, s_e. The transition table δ represents the DFA's transition diagram. Identical columns in δ have been combined, so the scanner uses the table *CharClass* to map an input character into a column index in δ. The roll back loop uses a stack of states to revert the scanner to its most recent accepting state.

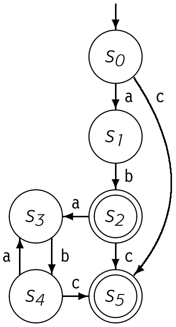
The function *NextChar* returns the next character in the input stream. A corresponding function, *RollBack*, moves the input stream back by one character (see Section 2.5.4).

When the scanning loop halts, *state* is s_e, the error state. The scanner backs up until it either finds an accepting state or it proves that none exists. In most languages, the amount of such roll back will be limited. We can construct REs that will require quadratic roll back on specific inputs. If the scanner must handle such an RE, an alternative implementation, such as the one described in the next subsection, should be used. In most programming languages, however, the amount of roll back should be small.

Avoiding Excess Roll Back

Some REs can cause the scanner in Fig. 2.12(a) to need quadratic roll back. The problem arises from the desire to have the scanner return the longest word that is a prefix of the input stream.

Consider the RE $ab \mid (ab)^*\, c$. The corresponding DFA, shown in the margin recognizes either ab or any number of occurrences of ab followed by a final c. On the input string `abababab c`, a scanner built from the DFA will read all the characters and return the entire string as a single word. If, however, the input is `abababab`, it must scan all of the characters before it can determine that the longest prefix is `ab`. On the next invocation, it will scan `ababab` to return `ab`. The third call will scan `abab` to return `ab`, and the final call will simply return `ab` without any roll back. In the worst case, roll back can create $O(n^2)$ behavior.



DFA for $ab \mid (ab)^*c$

The *maximal munch scanner* avoids this kind of pathological behavior by marking dead-end transitions as they are popped from the stack. Thus, over time, it records specific ⟨*state*,*input position*⟩ pairs that cannot lead to an accepting state. Inside the scanning loop, the code tests each ⟨*state*,*input position*⟩ pair and breaks out of the scanning loop whenever a failed transition is attempted.

```
state ← $s_0$;
lexeme ← "";
clear stack;
push(⟨$bad$, -1⟩);

while (state $\neq s_e$) do
    if Failed[state,InputPos] then
        ⟨state, InputPos⟩ ← pop();
        truncate lexeme;
        break;

    char ← Input[InputPos];
    lexeme ← lexeme + char;

    if state $\in S_A$ then
        clear stack;
        push(⟨$bad$, -1⟩);

    push(⟨state, InputPos⟩);

    col ← CharClass[char];
    state ← $\delta$[state, col];
    InputPos ← InputPos + 1;

while (state $\notin S_A$ and state $\neq$ $bad$) do
    if state $\neq s_e$ then
        Failed[state, InputPos] ← true;
    ⟨state, InputPos⟩ ← pop();
    if state $\neq$ $bad$ then
        truncate lexeme;

if state $\in S_A$
    then return TokenType[state];
    else return $invalid$;
```

■ **FIGURE 2.13** The Maximal Munch Scanner.

Fig. 2.13 shows the maximal munch scanner. The scanner keeps a global counter, `InputPos`, to record position in the input stream. It uses a bit-array, `Failed`, to record dead-end transitions. `Failed` has a row for each state and a column for each character in the input stream. When the scanner must roll back the input, it marks the appropriate transitions in `Failed` to prevent it from taking the same dead-end path on subsequent invocations.

The code in Fig. 2.13 runs on each call to the scanner. Before the first call to the scanner, both `Failed` and `InputPos` must be initialized. Every bit in `Failed` is set to `false`. `InputPos` is set to one.

Minor optimizations can reduce the size of `Failed`. For example, if the scanner uses a finite input buffer (see Section 2.5.4), the number of columns in `Failed` can be reduced to the size of the input buffer.

Most programming languages have simple enough microsyntax that this kind of quadratic roll back cannot occur. If, however, you are building a scanner for a language that has this problem, these techniques can avoid it for a small additional overhead per character.

2.5.2 Direct-Coded Scanners

To improve the performance of a table-driven scanner, we must reduce the cost of one or both of its basic actions: read a character and compute the next DFA transition. Direct-coded scanners reduce the cost of computing DFA transitions by replacing the explicit representation of the DFA's state and transition function with an implicit one. The implicit representation simplifies the two-step, table-lookup computation. It eliminates the memory references entailed in that computation and allows other specializations. The resulting scanner has the same functionality as the table-driven scanner, but with a lower overhead per character. A direct-coded scanner is no harder to generate than the equivalent table-driven scanner.

The table-driven scanner spends most of its time inside the central while loop; thus, the heart of a direct-coded scanner is an alternative implementation of that while loop. With some detail abstracted, that loop performs the following actions:

```
while (state $\neq s_e$) do
    char  ← NextChar();
    col   ← CharClass[char];
    state ← $\delta$[state,col];
```

Here, `state` explicitly represents the DFA's current state and δ is a two dimensional array that represents the DFA's transition function. Identical columns in δ have been combined (see Section 2.5.4). `CharClass` is a vector that maps an input character to a column index in δ.

Reducing the Overhead of Table Lookup

For each character, the table-driven scanner accesses two arrays: δ and `CharClass`. While both lookups take **O**(1) time, these table lookups have constant-cost overheads that a direct-coded scanner can avoid. To access the *i*th element of `CharClass`, the code must compute its address, given by

$$\texttt{@CharClass}_0 + i \times \texttt{w}$$

Detailed discussion of the code to compute an address for an array element starts on page 340 in Section 7.3.2.

where $\texttt{@CharClass}_0$ is a constant related to the memory address of *CharClass* and *w* is the size in bytes of an element of *CharClass*. The code then loads the column index found at that memory address.

Next, the scanner locates the state in δ. Because δ has two dimensions, the address calculation for δ`[state,col]` is more complex:

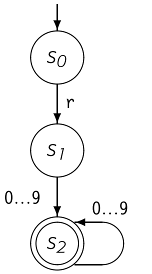
$$\texttt{@}\delta_0 + (\texttt{state} \times \textit{number of columns in } \delta + \texttt{col}) \times \texttt{w}$$

where $\texttt{@}\delta_0$ is a constant related to the starting address of δ in memory and `w` is the number of bytes per element of δ. Again, the scanner must issue a load operation to retrieve the data stored at this address.

Thus, the table-driven scanner computes two addresses and performs two loads for each input character. Some of the speed improvement in a direct-coded scanner comes from reducing this overhead.

Replacing the Table-Driven Scanner's While Loop

The table-driven scanner represents the DFA state and transition diagram explicitly, so that it can use the same code to implement each state. By contrast, a direct-coded scanner represents the state and transition diagram implicitly. It uses a distinct and customized code fragment to implement each state. It emulates state-to-state transitions by branching to the appropriate code fragments.



DFA for $r\,[0\ldots9]^+$

Fig. 2.14 shows a direct-coded scanner for $r\,[0\ldots9]^+$; the DFA is shown in the margin and the table-driven scanner appeared in Fig. 2.12. Execution begins at label $\texttt{s}_0$, which initializes the scanner and performs the actions for DFA state s_0.

Consider the code for state s_1. It reads a character, concatenates it onto the current lexeme, and pushes s_1 onto its internal stack. If *char* is a digit, it jumps to state s_2. Otherwise, it jumps to state s_{out}. The code performs no complicated address calculations. It refers to a tiny set of values—*char*, *lexeme*, and *state*—that can be kept in registers. The other states have equally simple implementations.

A scanner generator can directly emit code similar to that shown in Fig. 2.14. Each state has a couple of standard actions, followed by branching logic that implements the transitions out of the state. If some state has too

```
$s_0$:   clear stack;
         push(bad);
         char ← NextChar();
         lexeme ← char;
         push($s_0$);

         if (char = 'r')
            then goto $s_1$;
            else goto $s_{out}$;

$s_1$:   char ← NextChar();
         lexeme ← lexeme + char;
         push($s_1$);

         if ('0' ≤ char ≤ '9')
            then goto $s_2$;
            else goto $s_{out}$;

$s_2$:   char ← NextChar();
         lexeme ← lexeme + char;
         clear stack;
         push($s_2$);

         if '0' ≤ char ≤ '9'
            then goto $s_2$;
            else goto $s_{out}$;

$s_{out}$: state ← $s_e$;
         while (state ∉ $S_A$ and state ≠ bad) do
            state ← pop();
            if state ≠ bad then
                truncate lexeme;
                RollBack();
         end;

         if state ∈ $S_A$
            then return Type[state];
            else return invalid;
```

■ **FIGURE 2.14** A Direct-Coded Scanner for *r [0 ... 9]*$^+$.

many distinct outbound transitions, a clever implementation might build a small transition table and use a "computed" branch scheme—a table lookup into a table of labels. Unlike the table-driven scanner, the code changes for each set of REs. Since that code is generated directly from the REs, the difference should not matter to the compiler writer.

Of course, the generated code violates many of the precepts of structured programming. While small examples may be comprehensible, the code for a complex set of REs may be difficult for a human to follow. Again, since the code is generated, humans should not need to read or debug it. The additional speed obtained from direct coding makes it an attractive option, particularly since it entails no extra work for the compiler writer. Any extra work is pushed into the implementation of the scanner generator.

Code in the style of Fig. 2.14 is often called *spaghetti code* in recognition of its tangled control flow.

2.5.3 Hand-Coded Scanners

Generated scanners, whether table-driven or direct-coded, use a small, constant amount of time per character. Despite this fact, many compilers use hand-coded scanners. In an informal survey of commercial compiler groups, we found that a surprisingly large fraction used hand-coded scanners. Similarly, many of the popular open-source compilers rely on hand-coded scanners. For example, the `flex` scanner generator was ostensibly built to support

We suspect that hand-coded scanners persist for one simple reason: they are small, simple programs that can be fun to write.

the GCC project, but GCC 4.0 uses hand-coded scanners in several of its front ends.

The direct-coded scanner reduces the overhead of simulating the DFA; the hand-coded scanner offers a clever compiler writer additional opportunities to improve performance. The code along specific paths can be optimized.

For example, the scanner from Fig. 2.14 implements a DFA that has just one accepting state. Thus, the stack mechanism for tracking accepting states can be eliminated. The transitions $s_0 \rightarrow s_{out}$ and $s_1 \rightarrow s_{out}$ can be replaced with code that simply returns *invalid*.

Similarly, the interface between the scanner and the parser can be improved. The table-driven and direct-coded scanners for *r*[*0...9*]$^+$ return the lexeme as a character string. If the parser has a syntactic category *register name*, the scanner might return the actual register number rather than the string that contains it—avoiding that conversion in the parser and eliminating multiple concatenations in the scanner.

2.5.4 Practical Implementation Issues

This section addresses two practical issues that arise in building a scanner: handling the input stream in a fashion that allows both efficient character-by-character scanning and rollback; and compressing the transition table so that it requires less space.

Buffering the Input Stream

While character-by-character I/O leads to clean algorithms, the overhead of a function call per character is significant relative to the cost of simulating the DFA. To reduce the I/O cost per character, the compiler writer can use buffered I/O, where each read operation returns a longer string of characters in a buffer and the scanner indexes through the buffer. The scanner maintains a pointer into the buffer. Responsibility for filling the buffer, tracking the current location, and recognizing the end of file all fall to `NextChar`. These operations can be performed inline; they are often encoded in a macro to avoid cluttering the code with pointer dereferences and increments.

The cost of reading a full buffer has two components, a large fixed overhead and a small per-character cost. A buffer and pointer scheme amortizes the fixed costs of the read over many single-character fetches. Making the buffer larger reduces the number of times that the scanner incurs this cost and reduces the per-character overhead.

Using a buffer and pointer also leads to a simple and efficient implementation of the `RollBack` operation. To back up in the input stream, the scanner

```
NextChar() {
    Char ← Buffer[Input];
    if Char ≠ eof then
        Input ← (Input + 1) mod 2n;
        if (Input mod n = 0) then
            fill Buffer[Input : Input + n - 1];
            Fence ← (Input + n) mod 2n;
    return Char;
}
```

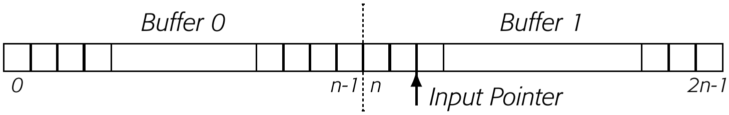
```
RollBack() {
    if (Input = Fence) then
        signal roll back error;
    else
        Input ← (Input - 1) mod 2n;
}

Initialize() {
    Input ← 0;
    Fence ← 0;
    fill Buffer[0 : n-1];
}
```

■ **FIGURE 2.15** Implementing *NextChar* and *RollBack*.

can simply decrement the input pointer. This scheme works as long as the scanner does not decrement the pointer beyond the start of the buffer. At that point, however, the scanner needs access to the prior contents of the buffer.

In practice, the compiler writer can bound the roll back distance that a scanner needs. With bounded roll back, the scanner can simply use two adjacent buffers and increment the pointer in a modulo fashion, as shown below:

Double buffering
A scheme that uses two input buffers in a modulo fashion to provide bounded roll back is often called *double buffering*.



To read a character, the scanner increments the pointer, modulo *2n*, and returns the character at that location. To roll back a character, the program decrements the input pointer, modulo *2n*. It must also manage the contents of the buffer, reading additional characters from the input stream as needed.

Both `NextChar` and `RollBack` have simple, efficient implementations, as shown in Fig. 2.15. Each execution of `NextChar` loads a character, increments the *Input* pointer, and tests whether or not to fill the buffer. Every `n` characters, it fills the buffer. The code is small enough to be included inline, perhaps generated from a macro. This scheme amortizes the cost of filling the buffer over `n` characters. By choosing a reasonable size for `n`, such as 2,048, 4,096, or more, the compiler writer can keep the I/O overhead low.

`RollBack` is even less expensive. It performs a test to ensure that the buffer contents are valid and then decrements the input pointer. Again, the imple-

δ	0	1	2	3	4	5	6	7	8	9	Other
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

(a) The Full Transition Table for *0 | [1... 9] [0... 9]**

δ	0	1…9	Other
s_0	s_1	s_2	s_e
s_1	s_e	s_e	s_e
s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e

(b) The Compressed Table

■ **FIGURE 2.16** Transition-Table Compression Example.

mentation is sufficiently simple to be expanded inline. `Initialize` simply provides a known and consistent starting state.

As a consequence of using finite buffers, `RollBack` has a limited history in the input stream. To keep it from decrementing the pointer beyond the start of that context, `NextChar` and `RollBack` cooperate. The pointer `Fence` always indicates the start of the valid context. `NextChar` sets `Fence` each time it fills a buffer. `RollBack` checks `Fence` each time it tries to decrement the `Input` pointer.

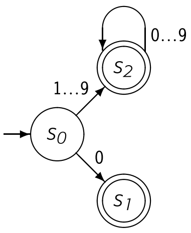
After a long series of `NextChar` operations, say, more than *n* of them where *n* is the buffer size, `RollBack` can always back up at least *n* characters. However, a sequence of calls to `NextChar` and `RollBack` that work forward and backward in the buffer can create a situation where the distance between `Input` and `Fence` is less than *n*. Larger values of *n* decrease the likelihood of this situation arising.

If *n* is chosen to make the I/O efficient, say 2,048 or 4,096 bytes for each half of the buffer, that should provide enough rollback for real programs. The amount of rollback required by any particular input is bounded by the longest sequence of whitespace-free characters. Few programs contain identifiers with more than 2,048 characters.

Compressing the Transition Table

The transition table for a DFA contains |states| · |Σ| entries. For a real programming language, both |states| and |Σ| can be large. When the table size grows larger than the size of the first-level cache, it may cause performance problems.

The transition table for a programming language scanner often contains identical columns. (If the DFA is minimal, its rows cannot be identical.) Consider, for example, the DFA for *0 | [1... 9] [0... 9]**, introduced in Section 2.2.2 and repeated in the margin. Fig. 2.16(a) shows the naive representation of the table. Panel (b) of that figure shows the same table, with



DFA for *0 | [1 ... 9] [0 ... 9]**

```
for i ← 1 to NumCols do
    MapTo[i] ← i

for i ← 1 to NumCols - 1 do
    if MapTo[i] = i then
        for j ← i + 1 to NumCols do
            if MapTo[j] = j then
                same ← true
                for k ← 1 to NumRows
                    if $\delta(k,i) \neq \delta(k,j)$ then
                        same ← false
                        break
                if same then
                    MapTo[j] ← i
```

■ **FIGURE 2.17** Finding Identical Columns in the Transition Table δ.

the columns for the characters 1 through 9 compressed into a single column. The code skeletons in Sections 2.12 and 2.13 create one more opportunity: the row for s_e cannot be referenced and, thus, need not be represented.

To use a compressed table, the scanner must map actual characters into columns in the transition table. A simple and efficient way to implement this translation is with a classification table that maps an input character to a column index. The fundamental loop in a scanner can be abstracted to the code shown on the left, below.

```
while (state ≠ sₑ) do                while (state ≠ sₑ) do
    char  ← NextChar();                  char  ← NextChar();
    state ← δ[state,char];               col   ← CharClass[char];
                                         state ← δ[state,col];
```

The corresponding code, with classification, is shown on the right. It adds one memory reference from `CharClass`; in return, the scanner can use a much smaller table representation.

Of course, the scanner generator must generate both the classifier and the compressed table. Fig. 2.17 shows the obvious algorithm to find identical columns. It assumes a transition table, δ with *NumCols* columns and *NumRows* rows. When it finishes, if *MapTo[i]* $= j$, for $i \neq j$, then rows i and j are identical and can be compressed to a single row. We leave the construction of the classifier table and the compressed version of δ as an exercise for the reader (see Exercise 2.13).

The algorithm, as shown, performs $\mathbf{O}(|\Sigma|^2)$ comparisons in the worst case. Each comparison costs $\mathbf{O}(|\text{states}|)$ time. We can reduce the quadratic term by keeping, for each column of δ, a population count of the nonerror states in the column. Since identical columns must have identical counts, the algorithm can construct an index set and radix sort it based on the population count. Then, it need only compare two columns if the radix sort puts them in the same category. This strategy should lead to multiple groups, each of which requires $\mathbf{O}(n^2)$ comparisons; however, each n should be smaller than $|\Sigma|$.

The count of nonerror states is a simple signature for the column. More complex, and possibly more expensive, signatures can be implemented. In the best case, they reduce the cost of the comparisons to $\mathbf{O}(1)$. There is, however, a direct tradeoff between the cost of the comparison and the cost of computing the signature—an issue for the careful implementor to consider.

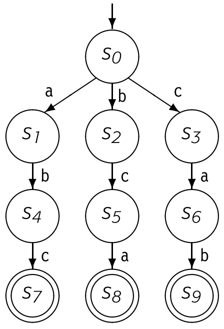
SECTION REVIEW

Automated techniques can easily build scanners from minimal DFAs. The scanner generator can adopt either a table-driven approach, with a generic skeleton scanner and language-specific tables, or a direct-coded scanner that threads together DFA states with explicit branches. In general, direct-coded scanners have a lower overhead per character than table-driven scanners.

Despite the fact that all DFA-based scanners have small constant costs per character, many compiler writers choose to hand code a scanner. This approach lends itself to careful implementation of the interfaces between the scanner and the I/O system and between the scanner and the parser.

REVIEW QUESTIONS

1. Given the DFA shown in the margin, complete the following:
   a. Build the transition table, based on the transition diagram and your character classifier.
   b. Write an equivalent direct-coded scanner.
2. An alternative implementation might use a DFA for (*a*|*b*|*c*) (*a*|*b*|*c*) (*a*|*b*|*c*), followed by a lookup in a table that contains the three words `abc`, `bca`, and `cab`.
   a. Sketch the DFA for this language.
   b. Contrast the cost of this approach with using the DFA from question one above.



DFA for Review Question 1

2.6 ADVANCED TOPICS

This section expands on the material in Section 2.4. The combination of Thompson's construction and the subset construction demonstrates that DFAs can express any computation that can be expressed as an RE. To complete the cycle and show that an RE can express any computation embodied by a DFA, we need a construction to create an RE that represents the set of words accepted by an arbitrary DFA. Section 2.6.1 sketches that construction, often called Kleene's algorithm.

Section 2.6.2 looks at an interesting subcase of the RE to DFA problem: closure-free REs. The DFA for a closure-free RE is acyclic. Section 2.6.2 sketches an algorithm to build such a DFA directly and incrementally.

Finally, Section 2.4.4 presented Hopcroft's algorithm for DFA minimization. Section 2.6.3 describes an alternative algorithm by Brzozowski that reuses the subset construction.

2.6.1 DFA to Regular Expression

The final step in the cycle of constructions, shown in Fig. 2.3, is to build an RE from a DFA. The combination of Thompson's construction and the subset construction provides a constructive proof that DFAs are at least as powerful as REs. This section presents Kleene's construction, which builds an RE to describe the set of strings accepted by an arbitrary DFA. This algorithm establishes that REs are at least as powerful as DFAs. Together, they show that REs and DFAs are equivalent.

Consider the transition diagram of a DFA as a graph with labeled edges. The problem of deriving an RE that describes the language accepted by the DFA corresponds to a path problem over the DFA's transition diagram. The set of strings in $L(\text{DFA})$ consists of the set of edge labels for every path from d_0 to d_j, $\forall\, d_j \in D_A$. For any DFA with a cyclic transition graph, the set of such paths is infinite. Fortunately, REs have the Kleene closure operator to handle this case and summarize the complete set of subpaths created by a cycle.

Fig. 2.18 shows one algorithm to compute this path expression. It assumes that the DFA has states numbered from 0 to $|D|-1$, with d_0 as the start state. It generates an expression that represents the labels along all paths between two nodes, for each pair of nodes in the transition diagram. As a final step, it combines the expressions for each path that leaves d_0 and reaches some accepting state, $d_j \in D_A$. It constructs the path expressions for all paths by iterating over i, j, and k.

```
for $i = 0$ to $|D|$ - 1 do
    for $j = 0$ to $|D|$ - 1 do
        $R^{-1}_{ij} \leftarrow \{\, a \mid \delta(d_i,a) = d_j \,\}$
        if ($i = j$) then
            $R^{-1}_{ij} \leftarrow R^{-1}_{ij} \mid \{\epsilon\}$

for $k = 0$ to $|D|$ - 1 do
    for $i = 0$ to $|D|$ - 1 do
        for $j = 0$ to $|D|$ - 1 do
            $R^{k}_{ij} \leftarrow R^{k-1}_{ik}\ (R^{k-1}_{kk})^*\ R^{k-1}_{kj} \mid R^{k-1}_{ij}$

$L \leftarrow |_{d_j \in D_A}\ R^{|D|-1}_{0j}$
```

■ **FIGURE 2.18** Deriving a Regular Expression from a DFA.

We use the notation R^k_{ij} to represent the RE that describes all paths from d_i to d_j that do not pass through a state numbered higher than d_k. Here, *through* means that the path both enters and leaves a state numbered higher than d_k. In a DFA with a transition $d_1 \rightarrow d_{16}$, the RE $R^2_{1,16}$ would be nonempty because the path enters d_{16} but does not pass through it.

Traditional statements of this algorithm assume that node names range from 1 to n, rather than from 0 to $n-1$. Thus, they place the direct paths in R^0_{ij}.

Initially, the algorithm places all of the direct paths from d_i to d_j in R^{-1}_{ij}. It adds $\{\epsilon\}$ to each expression where $i = j$. Over successive iterations, it builds up longer paths; it computes R^k_{ij} from R^{k-1}_{ij} by adding those paths that pass through d_k on their way from d_i to d_j. The algorithm computes this additional component as (1) the set of paths from d_i to d_k that pass through no state numbered higher than k-1, concatenated with (2) any paths from d_k to itself that pass through no state numbered higher than k-1, concatenated with (3) the set of paths from d_k to d_j that pass through no state numbered higher than k-1. The assignment in the inner loop

$$R^k_{ij} \leftarrow R^{k-1}_{ik}\ (R^{k-1}_{kk})^*\ R^{k-1}_{kj} \mid R^{k-1}_{ij}$$

captures those paths and uses alternation to add them to the RE for the paths from R^{k-1}_{ij}. In this way, each iteration of the inner loop adds the paths that pass through d_k to R^{k-1}_{ij} to form R^k_{ij}.

When the k loop terminates, the various R^k_{ij} expressions account for all paths through the graph. The final step computes the set of paths that start with d_0 and end in some accepting state, $d_j \in D_A$, as the alternation of the path expressions.

2.6.2 Closure-Free Regular Expressions

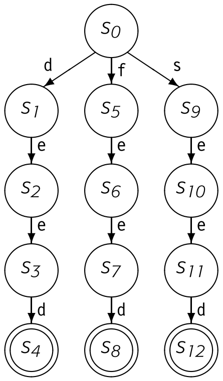
One subclass of regular languages that has practical application beyond scanning is the set of languages described by closure-free REs. Such REs have the form $w_1 \mid w_2 \mid w_3 \mid \ldots w_n$, where the individual words, w_i, are just concatenations of characters in the alphabet, Σ. These REs have the property that they produce DFAs with acyclic transition graphs.

These simple regular languages are of interest for two reasons. First, many pattern recognition problems can be described with a closure-free RE. Examples include words in a dictionary, URLs that should be filtered, and keys to a hash table. Second, the DFA for a closure-free RE can be built in a particularly efficient way.

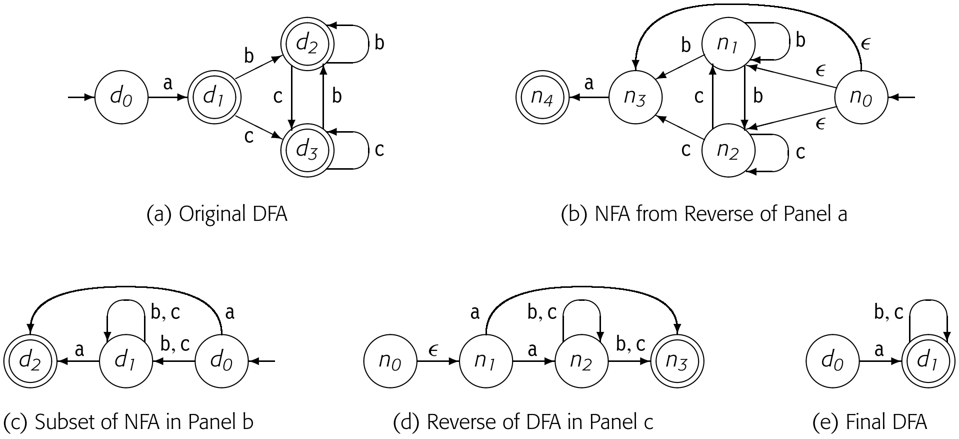
To build the DFA for a closure-free RE, begin with a start state s_0. To add a word to the existing DFA, the algorithm follows the path for the new word until it either exhausts the pattern or finds a transition to s_e. In the former case, it designates the final state for the new word as an accepting state. In the latter, it adds a path for the new word's remaining suffix. The resulting DFA can be encoded in tabular form or in direct-coded form (see Section 2.5.2). Either way, the recognizer uses constant time per character in the input stream.

In this algorithm, the cost of adding a new word to an existing DFA is proportional to the length of the new word. The algorithm also works incrementally; an application can easily add new words to a DFA that is in use. This property makes the acyclic DFA an interesting alternative for implementing a perfect hash function. For a small set of keys, this technique produces an efficient recognizer. As the number of states grows (in a direct-coded recognizer) or as key length grows (in a table-driven recognizer), the implementation may slow down due to cache-size constraints. At some point, the impact of cache misses will make an efficient implementation of a more traditional hash function more attractive than incremental construction of the acyclic DFA.

The DFAs produced in this way are not guaranteed to be minimal. Consider the acyclic DFA that it would produce for the REs *deed*, *feed*, and *seed*, shown in the margin. It has three distinct paths that each recognize the suffix *eed*. Clearly, those paths can be combined to reduce the number of states and transitions in the DFA. The algorithm will build a DFA that is minimal with regard to prefixes of words in the language, similar to those produced by the subset construction (see Section 2.4.3). A complete minimization would combine the suffixes, as well, to produce a smaller DFA.



DFA for *deed* | *feed* | *seed*



(a) Original DFA

(b) NFA from Reverse of Panel a

(c) Subset of NFA in Panel b

(d) Reverse of DFA in Panel c

(e) Final DFA

■ **FIGURE 2.19** Applying Brzozowski's Algorithm to the DFA for *a* (*b* | *c*)*.

2.6.3 An Alternative DFA Minimization Algorithm

The subset construction converted an NFA to a DFA by systematically eliminating ϵ-transitions and combining paths in the NFA's transition diagram. If we apply the subset construction to an NFA that has multiple paths from the start state for some prefix, the construction combines those paths into a single path. The resulting DFA has no duplicate prefixes. Brzozowski used this observation to devise an alternative minimization algorithm that directly constructs the minimal DFA from either an NFA or a DFA.

For NFAs built with Thompson's construction, reachability is not an issue. It can arise in minimizing an arbitrary NFA.

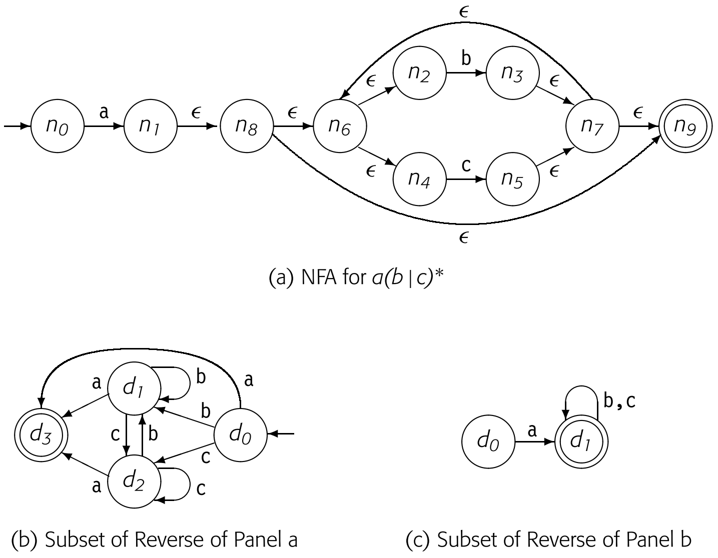
For an NFA *n*, let *reverse(n)* be the NFA obtained by reversing the direction of all the transitions, making the initial state into a final state, adding a new initial state, and connecting it to all of the states that were final states in *n*. Further, let *reachable(n)* be a function that returns the set of states and transitions in *n* that are reachable from its initial state. Finally, let *subset(n)* be the DFA produced by applying the subset construction to *n*.

Now, given an NFA *n*, the minimal equivalent DFA is just

reachable(subset(reverse(reachable(subset(reverse(n)))))).

The inner application of *subset* and *reverse* eliminates duplicate suffixes in the original NFA. Next, *reachable* discards any states and transitions that are no longer interesting. Finally, the outer application of the triple, *reachable*, *subset*, and *reverse*, eliminates any duplicate prefixes in the NFA. (Applying *reverse* to a DFA can produce an NFA.)

Fig. 2.19 shows the steps that the algorithm takes to minimize the DFA for *a* (*b* | *c*)* produced in the previous two subsections. Panel (a) repeats the



(a) NFA for $a(b\,|\,c)^*$

(b) Subset of Reverse of Panel a

(c) Subset of Reverse of Panel b

■ **FIGURE 2.20** Brzozowski's Algorithm Applied to the NFA from Fig. 2.5.

DFA from Fig. 2.7. Applying *reverse* to this DFA produces the NFA shown in panel (b). Next, the algorithm applies *subset* to this NFA, to produce the DFA shown in panel (c). *Reverse* applied to panel (c) yields the NFA in panel (d). Applying *subset* to this NFA produces the final DFA shown in panel (e), which is minimal. Note that reachability did not play a role in this example.

Brzozowski's algorithm can be expensive because of the potential for the subset construction to build an exponentially large set of states. Hopcroft's algorithm has a lower asymptotic complexity: $\mathbf{O}(|N|\,|\Sigma|\,\log_2(|N|))$, where N is the set of states in the input FA.

The tradeoff between the two algorithms is not straightforward. Studies of the running times of various FA minimization techniques suggest, however, that the actual running times depend on specific properties of the FA. In practice, Brzozowski's algorithm appears to perform reasonably well.

Furthermore, the implementation of Brzozowski's algorithm will almost certainly be simpler than that of Hopcroft's algorithm. Since Brzozowski's algorithm produces a DFA, it can be applied directly to the output of Thompson's construction, eliminating an extra application of the subset construction.

The first application of *subset* is, effectively, free.

Fig. 2.20 shows the steps that the algorithm takes when applied directly to the NFA that Thompson's construction built for $a\,(b\mid c)^*$. Panel (a) shows the original NFA. Panel (b) shows the DFA constructed by applying *reverse* and then *subset* to the NFA. Panel c shows the final DFA. The two reversed NFAs are left as an exercise for the reader.

Use in a Scanner Generator

Because Brzozowski's algorithm uses the subset construction, its first step combines all of the final states into a single representative state. As described in Section 2.4.5, the scanner needs a DFA where each final state maps to one syntactic category. We can modify Hopcroft's algorithm to maintain this map; Brzozowski's algorithm has no similar fix.

Applying Brzozowski's algorithm to an NFA will produce a DFA.

To use Brzozowski's algorithm in a scanner generator, the tool would need to build an NFA for each rule and apply Brzozowski's algorithm to the individual NFAs. It could then combine those DFAs with ϵ transitions and use the subset construction to produce a final DFA. As before, if this application of the subset construction merges final states, it must assign the new final state the syntactic category of the higher priority rule. The resulting DFA will not be minimal. It will, however, be smaller than the DFA produced without minimization.

2.7 SUMMARY AND PERSPECTIVE

The widespread use of REs for searching and scanning is one of the success stories of modern computer science. These ideas were developed as an early part of the theory of formal languages and automata. They are routinely applied in tools ranging from text editors to web filtering engines to compilers as a means of concisely specifying groups of strings that happen to be regular languages. Whenever a finite collection of words must be recognized, DFA-based recognizers deserve serious consideration.

The theory of REs and finite automata has developed techniques that allow the recognition of regular languages in time proportional to the length of the input stream. Techniques for automatic derivation of DFAs from REs and for DFA minimization have allowed the construction of robust tools that generate DFA-based recognizers. Both generated and hand-crafted scanners are used in well-respected modern compilers. In either case, a careful implementation should run in time proportional to the length of the input stream, with a small overhead per character.

CHAPTER NOTES

Originally, the separation of lexical analysis, or scanning, from syntax analysis, or parsing, was justified with an efficiency argument. Since the cost of scanning grows linearly with the number of characters, and the constant costs are low, pushing lexical analysis from the parser into a separate scanner lowered the cost of compiling. The advent of efficient parsing techniques weakened this argument, but the practice of building scanners persists because it provides a clean separation of concerns between lexical structure and syntactic structure.

Because scanner construction plays a small role in building an actual compiler, we have tried to keep this chapter brief. Thus, the chapter omits many theorems on regular languages and finite automata that the ambitious reader might enjoy. The many good texts on this subject can provide a much deeper treatment of finite automata and REs, and their many useful properties [206,241,327].

Kleene [235] established the equivalence of REs and FAs. Both the Kleene closure and the DFA to RE algorithm bear his name. McNaughton and Yamada showed one construction that relates REs to NFAs [270]. The construction shown in this chapter is patterned after Thompson's work [345]; to add regular-expression search to the QED text editor, he built a small compiler that translated an RE into native code for the IBM 7094. Johnson was the first to describe the application of these ideas to automate scanner construction [218]. The subset construction derives from Definition 11 in Rabin and Scott [302].

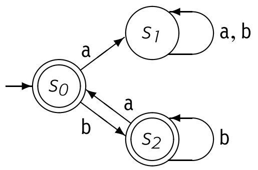
Hopcroft published his DFA minimization algorithm, presented in Section 2.4.4 in 1971 [205]. It has found application to a variety of problems, including detecting when two program variables always have the same value [23]. The alternative algorithm in Section 2.6.3 was published by Brzozowski in 1962 [66]. Several authors have compared DFA minimization techniques and their performance [340,355]. Several authors have looked at the construction and minimization of acyclic DFAs [122,356,357].

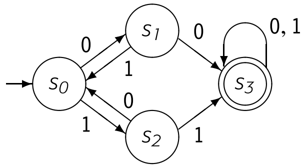
The maximal munch scanner is due to Reps [307]. The idea of generating code rather than tables, to produce a direct-coded scanner, appears to originate in work by Waite [353] and Heuring [201]. They report a factor of five improvement over table-driven implementations. Ngassam et al. describe experiments that characterize the speedups possible in hand-coded scanners [283]. Several authors have examined tradeoffs in scanner implementation. Jones [219] advocates direct coding but argues for a structured approach to control flow rather than the spaghetti code shown in Section 2.5.2. Brouwer et al. compare the speed of 12 different scanner implementations; they discovered a factor of 70 difference between the fastest and slowest implementations [65].

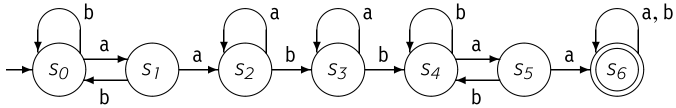
EXERCISES

Section 2.2

1. Describe informally the languages accepted by the following FAs:

a. 

b. 

c. 

2. Construct an FA accepting each of the following languages:

a. $\{w \in \{a, b\}^* \mid w$ starts with "*a*" and contains "*baba*" as a substring $\}$

b. $\{w \in \{0, 1\}^* \mid w$ contains "111" as a substring and does not contain "00" as a substring$\}$

c. $\{w \in \{a, b, c\}^* \mid$ the number of *a*'s modulo 2 in a word in *w* equal to the number of *b*'s modulo 3 in the same word$\}$

3. Create FAs to recognize (a) strings that represent complex numbers and (b) strings that represent decimal numbers written in scientific notation.

Section 2.3

4. Different programming languages use different notations to represent integers. Construct a regular expression for each one of the following:

a. Nonnegative integers in C represented in bases 10 and 16.

b. Nonnegative integers in VHDL that may include underscores (an underscore cannot occur as the first or last character).

c. Currency, in dollars, represented as a positive decimal number rounded to the nearest one-hundredth. Such numbers begin with the character $, have commas separating each group of three digits to the left of the decimal point, and end with two digits to the right of the decimal point, for example, $8,937.43 and $7,777,777.77.

5. Write a regular expression for each of the following languages:

   a. Given an alphabet $\Sigma = \{0, 1\}$, L is the set of all strings of alternating pairs of 0s and pairs of 1s.

   b. Given an alphabet $\Sigma = \{0, 1\}$, L is the set of all strings of 0s and 1s that contain an even number of 0s or an even number of 1s.

   c. Given the lowercase English alphabet, L is the set of all strings in which the letters appear in ascending lexicographical order.

6. Write a regular expression to describe each of the following programming language constructs:

   a. Any sequence of tabs and blanks (e.g., *whitespace*)

   b. Comments in the programming language C

   c. String constants (without escape characters)

   d. Floating-point numbers

Section 2.4

7. Consider the three regular expressions:

   $(ab \mid ac)^*$

   $(0 \mid 1)^*\ 1100\ \ 1^*$

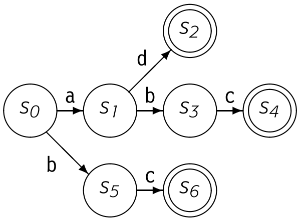
   $(01 \mid 10 \mid 00)^*\ \ 11$

   a. Use Thompson's construction to construct an NFA for each RE.

   b. Convert the NFAs to DFAs.

   c. Minimize the DFAs.

8. Apply Hopcroft's minimization algorithm to the DFA shown below.



9. Show that the set of regular languages is closed under intersection.

Section 2.5

10. Construct a DFA for each of the following C language constructs, and then build the corresponding table for a table-driven implementation for each of them:

    a. Integer constants

    b. Identifiers

    c. Comments

11. For each DFA in the previous exercise, write a direct-coded scanner.

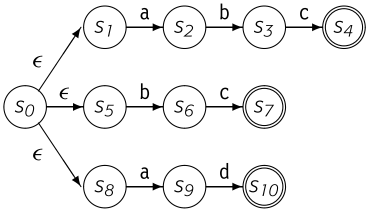
12. This chapter describes several ways to implement a DFA. Another alternative would use mutually recursive functions to implement a scanner. Discuss the advantages and disadvantages of such an implementation.

13. Fig. 2.17 shows an algorithm that discovers identical columns in the transition function, δ. Give an algorithm that constructs the character classifier, a map from characters to column numbers, and the reduced form of δ.

    Assume that column i of δ corresponds to the ith character in the alphabet, Σ, denoted Σ_i.

Section 2.6

14. Apply Brzozowski's algorithm to the following NFA:



Chapter 3

Parsers

ABSTRACT

The parser's task is to determine if the input program, represented by the stream of classified words produced by the scanner, is a valid sentence in the programming language. To do so, the parser attempts to build a derivation for the input program, in a grammar that describes the programming language.

This chapter introduces context-free grammars to specify programming language syntax. It develops both top-down and bottom-up parsing techniques. It describes ways to automate the construction of both of these types of parsers. Finally, it explores a number of practical issues that arise in parser construction.

KEYWORDS

Parsing, Grammar, LL(1), LR(1), Recursive Descent

3.1 INTRODUCTION

Parsing is the second stage of the compiler's front end. The parser works with the program as tokenized by the scanner; it sees a stream of words annotated with their syntactic categories (analogous to their parts of speech). The parser derives a syntactic structure for the program, fitting the words into a grammatical model of the source programming language. If the parser determines that the input stream is a valid program, it builds a concrete model of the program, an intermediate representation, for use by the rest of the compiler. If the parser finds errors, it reports both the problem and its location to the user.

Parsing and scanning are similar. Like scanning, parsing has been studied extensively; modern-day parsers build on that theoretical basis. Speed matters; all of the techniques that we will study take time proportional to the size of the program and its representation. Low-level detail affects performance; the same implementation tradeoffs arise in parsing as in scanning. The techniques in this chapter are amenable to implementation as table-driven parsers, direct-coded parsers, and hand-coded parsers. Unlike scanners, where hand-coding is common, tool-generated parsers are more common than hand-coded parsers.

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00009-7

Conceptual Roadmap

The parser's primary task is to determine whether or not the input program is a syntactically valid sentence in the source language. Before we can build parsers to answer this question, we need both a formal mechanism to specify the source language's syntax and a systematic way to determine membership in this formally specified language. By restricting the form of the source language to a set of languages called context-free languages, we can ensure an efficient algorithm to answer the membership question. Context-free grammars (CFGs) are the notation used to specify context-free languages.

Many algorithms have been proposed to answer the membership question for CFGs. This chapter examines two different approaches to the problem: top-down parsers and bottom-up parsers. The two styles of parsers differ significantly in their approach and implementation. Both styles, however, handle a large class of grammars that includes most of the programming-language constructs that occur in modern languages. Equally important, tools are widely available to assist the compiler writer in building either a top-down parser or a bottom-up parser. This chapter explores both the parsing techniques and the methods used to automate parser construction.

Overview

Parsing
Given a stream *s* of words and a grammar *G*, find a derivation in *G* that produces *s*.

A compiler's parser has the primary responsibility for recognizing syntax—that is, for determining if the program being compiled is a valid sentence in the syntactic model of the programming language. That model is expressed as a formal grammar *G*; if some string of words *s* is in the language defined by *G* we say that *G derives s*. For a stream of words *s* and a grammar *G*, the parser tries to build a constructive proof that *s* can be derived in *G*—a process called *parsing*.

Recursive-descent parsers
Recursive-descent parsers are hand-coded, top-down parsers. They can be compact and efficient.

LL(1) parsers
LL(1) parsers are table-driven, top-down parsers. They recognize a class of grammars that include most interesting programmming language features.

LR(1) parsers
LR(1) parsers are table-driven, bottom-up parsers. They recognize a larger class of grammars than do the LL(1) parsers.

Parsing algorithms fall into two general categories. Top-down parsers try to match the input stream against the productions of the grammar by predicting the next word (at each point). For a limited class of grammars, such prediction can be both accurate and efficient. Section 3.3 explores the details of how top-down parsers work and the techniques used to create them. It explores the construction of both recursive-descent and LL(1) parsers. Bottom-up parsers work from low-level detail—the actual sequence of words—and accumulate context until the derivation is apparent. Again, there exists a restricted class of grammars for which we can generate efficient bottom-up parsers. Section 3.4 examines one particular kind of bottom-up parser, a table-driven LR(1) parser, along with the techniques used generate these highly efficient parsers. The final sections explore a number of practical issues that arise in parser construction.

A Few Words About Time

Designing, building, and using a parser spans the entire continuum of compilation. At design time, the compiler writer chooses a parsing method and toolset. She then creates a CFG for the source language in the input format of the tools that she has chosen.

At build time, the compiler writer's development tools construct an executable parser. In a hand-written parser, the code is compiled directly. In a generated parser, the process invokes a parser generator to build the parser from the CFG and its annotations; that code is then compiled to create the executable parser.

Finally, at compile-time, the parser analyzes the tokenized version of the source program. It maps the stream of classified words into the CFG and identifies mismatches, if any, between the source program and the CFG. If the input program is correct, it generates an intermediate representation for the rest of the compiler to use. If the input contains errors, the parser reports them back to the programmer.

3.2 EXPRESSING SYNTAX

The task of the parser is to determine whether or not some stream of words fits into the syntax of the parser's intended source language. Implicit in this description is the notion that we can describe syntax and check it; in practice, we need a notation to describe the syntax of languages that people might use to program computers. In Chapter 2, we worked with one such notation, regular expressions. RE's provide a concise notation for describing limited kinds of syntax. RE descriptions lead to efficient recognizers. Unfortunately, REs lack the power to describe the full syntax of most programming languages.

For most programming languages, syntax is expressed with a CFG. This section introduces CFGs and explores their use in syntax-checking. It shows how to encode meaning into syntax and structure. Finally, it introduces the ideas that underlie the efficient parsing techniques described in later sections.

3.2.1 Why Not Use Regular Expressions?

To motivate CFGs, consider the problem of recognizing algebraic expressions over names and the operators +, -, ×, and ÷. We can define "name" as any string that matches the RE $[a\ldots z]\,([a\ldots z] \mid [0\ldots 9])^*$, a simplified, lowercase version of an Algol identifier. Now, we can define an expression as follows:

$$[a\ldots z]\,([a\ldots z]\mid[0\ldots 9])^*\ \ ((+\mid-\mid\times\mid\div)\ \ [a\ldots z]\,([a\ldots z]\mid[0\ldots 9])^*\,)^*$$

This RE matches “a + b × c” and “e ÷ f × g”. Nothing about the RE suggests a notion of operator precedence; in “a + b × c,” which operator executes first, the + or the × ? The standard rule from algebra suggests × and ÷ have precedence over + and -. To enforce other evaluation orders, normal algebraic notation includes parentheses.

We underline $\underline{(}$ and $\underline{)}$ to make them visually distinct from the (and) used for grouping in REs.

Can we add parentheses to the RE for expressions in the places where they would be legal? An expression can start with a $\underline{(}$, so the RE needs an optional initial $\underline{(}$. Similarly, it needs an optional final $\underline{)}$.

$$\begin{gathered}(\,\underline{(}\mid\epsilon\,)\,[a\ldots z]\,(\,[a\ldots z]\mid[0\ldots 9]\,)^*\\ (\,(\,+\mid-\mid\times\mid\div\,)\,[a\ldots z]\,(\,[a\ldots z]\mid[0\ldots 9])^*\,)^*\,(\,\underline{)}\mid\epsilon\,)\end{gathered}$$

This RE can produce an expression enclosed in parentheses, but not one with internal parentheses to denote precedence. The internal instances of $\underline{(}$ all occur before a name while the internal instances of $\underline{)}$ all occur after a name, which suggests the following RE:

$$\begin{gathered}\underline{(}^*\ [a\ldots z]\,(\,[a\ldots z]\mid[0\ldots 9]\,)^*\ \underline{)}^*\\ (\,(\,+\mid-\mid\times\mid\div\,)\ \underline{(}^*\ [a\ldots z]\,(\,[a\ldots z]\mid[0\ldots 9])^*\ \underline{)}^*\,)^*\end{gathered}$$

This RE matches both “a + b × c” and “(a + b) × c.” It will match any correctly parenthesized expression over names and the four operators in the RE. Unfortunately, it also matches many syntactically incorrect expressions, such as “a + (b × c” and “a + b) × c).” In fact, we cannot write an RE that will match all expressions with balanced parentheses. (Paired constructs, such as begin and end or then and else, play an important role in most programming languages.)

$a^n\ b^n$ is a classic example used to teach the pumping lemma in automata theory.

The inability to match brackets, be they $\underline{(}\;\;\underline{)}$, $\underline{\{}\;\;\underline{\}}$, or begin and end is a fundamental limitation of REs; the corresponding recognizers cannot count because they have only a finite set of states. The language $\underline{(}^n\ \underline{)}^n$ is not regular. In principle, DFAs cannot count. While they work well for microsyntax, they are not suitable to describe some important programming language features.

3.2.2 Context-Free Grammars

Context-free grammar
For a language L, its CFG defines the set of strings of symbols that are valid in L.

To describe programming language syntax, we need a more powerful notation than regular expressions that still leads to efficient recognizers. The

BACKUS-NAUR FORM

The traditional notation used by computer scientists to represent a context-free grammar is called *Backus-Naur form*, or BNF. In BNF, nonterminal symbols were wrapped in angle brackets, as in ⟨`SheepNoise`⟩. Terminal symbols were underlined, as in `baa`. The symbol `::=` meant "derives," and the symbol | meant "also derives." Writing the sheep noise grammar in BNF yields:

```
⟨SheepNoise⟩  ::=  baa ⟨SheepNoise⟩
                |  baa
```

BNF had its origins in the late 1950s and early 1960s [282]. The syntactic conventions of angle brackets, underlining, `::=`, and | arose from the limited typographic options available to compiler writers at the time. (For example, see David Gries' book *Compiler Construction for Digital Computers* [181].) Throughout this book, we use a typographically updated form of BNF. Nonterminal symbols are written in a *slanted sans-serif font*. Terminal symbols are set in a `typewriter font`. When the change in font is difficult to see, as with `)`, we underline the character as well, `)`. We use the symbol → for "derives" and | for "also derives."

traditional solution is to use a CFG. Fortunately, large subclasses of the CFGs have the property that they lead to efficient recognizers.

Production
Each rule in a CFG is called a *production*.

A CFG, *G*, is a set of rules, or *productions*, that describe how to form sentences. The collection of sentences that derive from *G* is called the *language defined by G*, denoted *L*(*G*). The set of languages defined by all possible CFGs is called the set of context-free languages. An example may help. Consider the following grammar, which we call *SN*:

Sentence
a string of symbols that can be derived from the rules of a grammar

```
1   SheepNoise  →  baa SheepNoise
2               |  baa
```

The first rule, or production, reads "*SheepNoise* can derive the word `baa` followed by another *SheepNoise*." Here *SheepNoise* is a syntactic variable representing the set of strings that can be derived from the grammar. We call such a syntactic variable a *nonterminal symbol*. Each word in the language, such as `baa`, is a *terminal symbol* in the grammar. The second rule reads "*SheepNoise* can also derive the word `baa`."

Nonterminal symbol
a syntactic variable used in a grammar's productions

Terminal symbol
a word that can occur in a sentence

A word consists of a lexeme and its syntactic category. Words are represented in a grammar by their syntactic category.

To understand the relationship between the *SN* grammar and *L*(*SN*), we need to specify how to apply rules in *SN* to derive sentences in *L*(*SN*). To begin, we must identify the start symbol of *SN*. It represents the set of all strings

CONTEXT-FREE GRAMMARS

Formally, a context-free grammar G is a quadruple (T, NT, S, P) where:

T is the set of terminal symbols, or words, in the language $L(G)$. Terminal symbols correspond to syntactic categories returned by the scanner.

NT is the set of nonterminal symbols. They are syntactic variables introduced to provide abstraction and structure in the productions of G.

S is a nonterminal symbol designated as the *start symbol* or *goal symbol* of the grammar. S represents the set of sentences in $L(G)$.

P is the set of productions or rewrite rules in G. Each rule in P has the form $NT \rightarrow (T \cup NT)^+$; that is, it replaces a single nonterminal with a string of one or more grammar symbols.

The sets *T* and *NT* can be derived directly from the productions. The start symbol may be unambiguous, as in the *SheepNoise* grammar, or it may not be obvious, as in the following grammar:

$$\begin{array}{lcl@{\qquad\qquad}lcl} \textit{Paren} & \rightarrow & \underline{(}\ \textit{Bracket}\ \underline{)} & \textit{Bracket} & \rightarrow & \underline{[}\ \textit{Paren}\ \underline{]} \\ & | & \underline{(}\ \underline{)} & & | & \underline{[}\ \underline{]} \end{array}$$

In this case, the choice of start symbol determines the outer brackets. Starting with *Paren* ensures that the outer brackets are parentheses, while starting with *Bracket* forces square brackets on the outside. To allow either, the compiler writer can add a new nonterminal symbol *S* and the two productions *S* → *Paren* | *Bracket*.

Some tools that manipulate grammars require that *S* not appear on the right-hand side of any production, which makes *S* easy to discover.

in $L(SN)$. As such, it cannot be one of the words in the language. Instead, it must be one of the nonterminal symbols introduced to add structure and abstraction to the language. Since SN has only one nonterminal, *SheepNoise* is the start symbol.

Derivation
a sequence of rewriting steps that begins with the grammar's start symbol and ends with a sentence in the language

To derive a sentence, we start with a prototype string that contains just the start symbol. We then repeat the following process: (1) pick a nonterminal symbol, α, in the prototype string; (2) choose a grammar rule, $\alpha \rightarrow \beta$; and (3) rewrite α with β. When the prototype string contains only terminal symbols, the derivation halts. The prototype string has been rewritten into a sentence in the language.

Sentential form
a string of symbols that occurs as one step in a valid derivation

At each point in the derivation, the prototype string consists of a sequence of terminal and nonterminal symbols. When such a string occurs as a step in a valid derivation, it is a *sentential form*. Any sentential form can be derived from the start symbol in zero or more steps. Similarly, from any sentential form we can derive a valid sentence in zero or more steps. Thus, if we begin with *SheepNoise* and apply successive rewrites using the two rules, at each step in the process the string is a sentential form. When we have reached the point where the string contains only terminal symbols, the string is a sentence in $L(SN)$.

To derive a sentence in *SN*, we start with the string that consists of one symbol, *SheepNoise*. We can rewrite *SheepNoise* with either rule 1 or rule 2. If we rewrite *SheepNoise* with rule 2, the string becomes `baa` and has no further opportunities for rewriting. The rewrite shows that `baa` is a valid sentence in $L(SN)$. The other choice, rewriting the initial string with rule 1, leads to a string with two symbols: `baa` *SheepNoise*. This string has one remaining nonterminal; rewriting it with rule 2 leads to the string `baa baa`, which is a sentence in $L(SN)$. We can write these derivations in tabular form:

Rule	Sentential Form
	SheepNoise
2	`baa`

Rewrite with Rule 2

Rule	Sentential Form
	SheepNoise
1	`baa` *SheepNoise*
2	`baa baa`

Rewrite with Rule 1 Then 2

As a notational convenience, we will use $\rightarrow^{+}$ to mean "derives in one or more steps." Thus, *SheepNoise* $\rightarrow^{+}$ `baa` and *SheepNoise* $\rightarrow^{+}$ `baa baa`.

Rule 1 lengthens the sentential form while rule 2 eliminates the nonterminal *SheepNoise*. (The sentential form never contains more than one instance of *SheepNoise*.) All derivations in *SN* begin with zero or more applications of rule 1, and end with rule 2. Applying rule 1 k times followed by rule 2 generates a string with $k + 1$ `baa`s.

Notice that $L(SN)$ can be specified with an RE as $(\texttt{baa})^{+}$. $L(SN)$ is a *regular language*—a member of the subset of CFGs that can be specified by an RE (see "Classes of Context-Free Grammars" on page 135).

3.2.3 More Complex Examples

The *SheepNoise* grammar is too simple to exhibit the power and complexity of CFGs. Instead, let us revisit the example that showed the shortcomings of REs: the language of expressions with parentheses.

1	*Expr*	→	(*Expr*)	4	*Op*	→	+
2		|	*Expr Op Expr*	5		|	-
3		|	`name`	6		|	×
				7		|	÷

Parse tree
a graph that represents a derivation; also called a *syntax tree*

Beginning with the start symbol, *Expr*, the grammar generates two kinds of subterms: parenthesized subterms, with rule 1, or plain subterms, with rule 2. To generate the sentence `(a+b)×c`, we can use the rewrite sequence (2,3,6,1,2,3,4,3) as shown in Fig. 3.1(a). Panel (b) shows the *parse tree*, a graphical representation of the derivation.

This simple CFG for expressions cannot generate a sentence with unbalanced or improperly nested parentheses. Only rule 1 can generate a `(`. The same rule also generates the matching `)`. Thus, it cannot generate strings such as `a+(b×c` or `a+b)×c)`, and a parser built from the grammar will not accept such strings. (The best RE in Section 3.2.1 matched both of these strings.) Clearly, CFGs have the ability to specify constructs that REs do not.

Rightmost derivation
a derivation that rewrites, at each step, the rightmost nonterminal

Leftmost derivation
a derivation that rewrites, at each step, the leftmost nonterminal

The derivation of `(a+b)×c` in Fig. 3.1(a) rewrote, at each step, the rightmost remaining nonterminal symbol. This systematic behavior was a choice; other choices are possible. One obvious alternative is to rewrite the leftmost nonterminal at each step. Using leftmost choices would produce a different derivation sequence for the same sentence. Panels (c) and (d) in Fig. 3.1 show the leftmost derivation.

The leftmost and rightmost derivations use the same set of rules; they apply those rules in a different order. Because a syntax tree represents the rules applied, but not the order of their application, the parse trees for the two derivations are identical.

Ambiguity
A grammar G is *ambiguous* if some sentence in $L(G)$ has more than one rightmost (or leftmost) derivation.

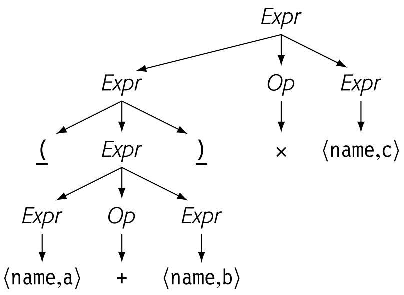
Alternatively, a grammar G is ambiguous if some sentence in $L(G)$ has more than one parse tree.

From the compiler's perspective, it is important that each sentence in the language defined by a CFG has a unique rightmost (or leftmost) derivation. If multiple rightmost (or leftmost) derivations exist for some sentence, then, at some point in the derivation, multiple distinct rewrites of the rightmost (or leftmost) nonterminal lead to the same sentence. A grammar in which multiple rightmost (or leftmost) derivations exist for a sentence is called an *ambiguous* grammar. An ambiguous grammar can produce multiple derivations and multiple parse trees.

For an ambiguous grammar, some inputs can produce multiple derivations and multiple meanings. A program must have a single meaning; otherwise, the compiler cannot know what code to generate. Thus, *ambiguity is a bad property in a programming language*.

Rule	Sentential Form
	Expr
2	*Expr Op Expr*
3	*Expr Op* `name`
6	*Expr* × `name`
1	(*Expr*) × `name`
2	(*Expr Op Expr*) × `name`
3	(*Expr Op* `name`) × `name`
4	(*Expr* + `name`) × `name`
3	(`name` + `name`) × `name`

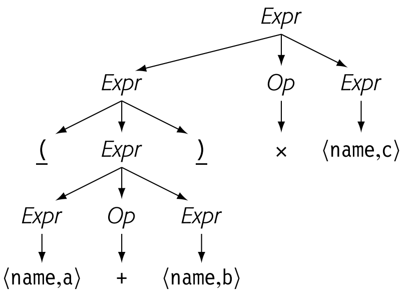
(a) Rightmost Derivation of (a + b) × c



(b) Corresponding Rightmost Parse Tree

Rule	Sentential Form
	Expr
2	*Expr Op Expr*
1	(*Expr*) *Op Expr*
2	(*Expr Op Expr*) *Op Expr*
3	(`name` *Op Expr*) *Op Expr*
4	(`name` + *Expr*) *Op Expr*
3	(`name` + `name`) *Op Expr*
6	(`name` + `name`) x *Expr*
3	(`name` + `name`) x `name`

(c) Leftmost Derivation of (a + b) × c



(d) Corresponding Leftmost Parse Tree

■ **FIGURE 3.1** Derivations of (a + b) × c.

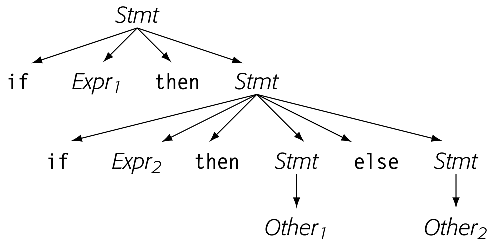
The classic example of an ambiguous grammar for a programming language construct is the `if–then–else` construct of many Algol-like languages. The straightforward grammar for `if–then–else` might be

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *Stmt* `else` *Stmt*
3		|	*Other*

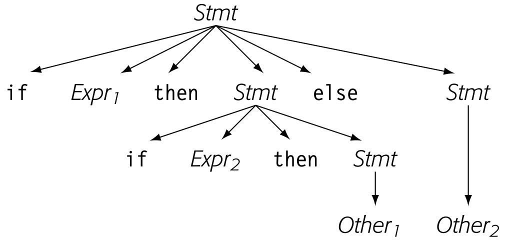
where *Other* represents all the statements that can appear in a list of statements, other than an `if–then` or an `if–then–else`. The grammar shows that the `else` clause is optional. Unfortunately, the sentence

`if` $Expr_1$ `then if` $Expr_2$ `then` $Other_1$ `else` $Other_2$

has two distinct rightmost derivations. The difference between them is simple. The first derivation has *Other*$_2$ controlled by the inner `if`, so *Other*$_2$ executes when *Expr*$_1$ is true and *Expr*$_2$ is false:



The second derivation associates the `else` clause with the first `if`, so *Other*$_2$ executes when *Expr*$_1$ is false, independent of the value of *Expr*$_2$:



These two derivations produce two distinctly different meanings.

To remove this ambiguity, we must modify the grammar to encode a rule that determines which `if` controls an `else`. The classic solution to the `if`–`then`–`else` ambiguity is to rewrite the grammar as follows:

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *WithElse* `else` *Stmt*
3		|	*Other*
4	*WithElse*	→	`if` *Expr* `then` *WithElse* `else` *WithElse*
5		|	*Other*

This solution restricts the set of statements that can occur in the `then` part of an `if`–`then`–`else` construct. It accepts the same set of sentences as the original grammar, but ensures that each `else` has an unambiguous match to a specific `if`. It encodes into the grammar a simple rule—bind each `else` to the innermost unmatched `if`. It has only one rightmost derivation for the example, which matches the first parse tree.

Rule	Sentential Form
	Stmt
1	`if` *Expr* `then` *Stmt*
2	`if` *Expr* `then` `if` *Expr* `then` *WithElse* `else` *Stmt*
3	`if` *Expr* `then` `if` *Expr* `then` *WithElse* `else` *Other*
5	`if` *Expr* `then` `if` *Expr* `then` *Other* `else` *Other*

The rewritten grammar eliminates the ambiguity.

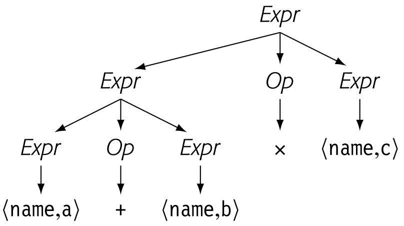
The `if–then–else` ambiguity arises from a shortcoming in the original grammar. The solution resolves the ambiguity by imposing a rule that the programmer can easily remember. In Section 3.5.3, we will look at other kinds of ambiguity and systematic ways of handling them.

3.2.4 Encoding Meaning into Structure

The `if–then–else` ambiguity points out the relationship between meaning and grammatical structure. However, ambiguity is not the only situation where meaning and grammatical structure interact. Consider the parse tree that would be built from a rightmost derivation of the simple expression `a + b × c`.

Rule	Sentential Form
	Expr
2	*Expr* *Op* *Expr*
3	*Expr* *Op* `name`
6	*Expr* × `name`
2	*Expr* *Op* *Expr* × `name`
3	*Expr* *Op* `name` × `name`
4	*Expr* + `name` × `name`
3	`name` + `name` × `name`

Derivation of a + b × c



Corresponding Parse Tree

One natural way to evaluate the expression is with a simple postorder treewalk. It would first compute `a + b` and then multiply that result by `c` to produce the result `(a + b) × c`. This evaluation order contradicts the classic rules of algebraic precedence, which would evaluate it as `a + (b × c)`. Since the ultimate point of parsing the expression is to produce code that will implement it, the expression grammar should have the property that it builds a tree whose "natural" treewalk evaluation produces the correct result.

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	num
9		|	name

■ **FIGURE 3.2** The Classic Expression Grammar.

The real problem lies in the structure of the grammar on page 92. It treats all the arithmetic operators in the same way, ignoring precedence. A rightmost derivation of a + b × c generates a different parse tree than does a leftmost derivation of the same string. The grammar is ambiguous.

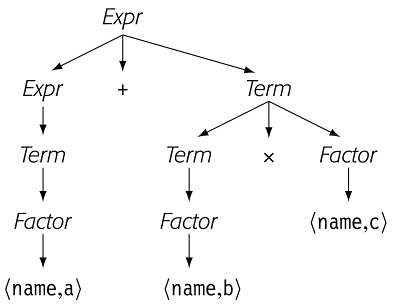
The example in Fig. 3.1 showed a string with parentheses. The parentheses forced the leftmost and rightmost derivations into the same parse tree. That extra production in the grammar added a level to the parse tree that, in turn, forces the same evaluation order independent of derivation order.

We can use this effect to encode levels of precedence into the grammar. First, we must decide how many levels of precedence are required. The simple expression grammar needs three precedence levels: highest precedence for ( ), medium precedence for × and ÷, and lowest precedence for + and -. Next, we group the operators at distinct levels and use a nonterminal to isolate that part of the grammar. Fig. 3.2 shows the resulting grammar; it adds a start symbol, *Goal* and a production for the terminal symbol num.

In the classic expression grammar, *Expr* forms a level for + and -, *Term* forms alevel for × and ÷, and *Factor* forms a level for ( ). The modified grammar derives a parse tree for a + b × c that models standard algebraic precedence.

Rule	Sentential Form
0	*Expr*
1	*Expr* + *Term*
4	*Expr* + *Term* × *Factor*
9	*Expr* + *Term* × name
6	*Expr* + *Factor* × name
9	*Expr* + name × name
3	*Term* + name × name
6	*Factor* + name × name
9	name + name × name

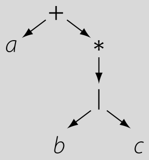
Rightmost Derivation of a + b × c



Corresponding Parse Tree

REPRESENTING THE PRECEDENCE OF OPERATORS

Thompson's construction must apply its three transformations in an order that is consistent with the precedence of the operators in the regular expression. To represent that order, an implementation of Thompson's construction can build a tree that represents the regular expression and its internal precedence. The RE $a(b|c)^*$ produces the following tree:



where + represents concatenation, | represents alternation, and * represents closure. The parentheses are folded into the structure of the tree and, thus, have no explicit representation.

The construction applies the individual transformations in a postorder walk over the tree. Since transformations correspond to operations, the postorder walk builds the following sequence of NFAs: a, b, c, $b|c$, $(b|c)^*$, and, finally, $a(b|c)^*$. Section 5.3 discusses a mechanism to build expression trees.

A postorder treewalk over this parse tree will first evaluate `b × c` and then add the result to `a`. The grammar enforces the standard rules of arithmetic precedence. This grammar is unambiguous; the leftmost derivation produces the same parse tree.

The changes affect derivation lengths and parse tree sizes. The new nonterminals that enforce precedence add steps to the derivation and interior nodes to the tree. At the same time, moving the operators inline eliminated one production and one node per operator.

Other operations require high precedence. For example, array subscripts should be applied before standard arithmetic operations. This ensures, for example, that `a + b[i]` evaluates `b[i]` to a value before adding it to `a`, as opposed to treating `i` as a subscript on some array whose location is computed as `a + b`. Similarly, operations that change the type of a value, known as *type casts* in languages such as C or JAVA, have higher precedence than arithmetic operations but lower precedence than parentheses or subscript operations.

If the language allows assignment inside expressions, the assignment operator should have low precedence. This ensures that the code completely evaluates both the left-hand side and the right-hand side of the assignment before performing the assignment. If assignment (←) had the same precedence as addition, for example, a left-to-right evaluation of `a ← b + c` would assign `b`'s value to `a` rather than evaluating `b + c` and then assigning that result to `a`.

3.2.5 Discovering a Derivation for an Input String

We have seen how to use a CFG G as a rewrite system to generate sentences that are in $L(G)$. By contrast, a parser takes a given input string, alleged to be in $L(G)$, and finds a derivation. The process of constructing a derivation from a specific input sentence is called *parsing*.

The parser sees the program as it emerges incrementally from the scanner: a stream of words annotated with their syntactic categories. Thus, the parser would see `a + b × c` as ⟨`name,a`⟩ + ⟨`name,b`⟩ × ⟨`name,c`⟩. As output, the parser needs to produce either a derivation of the input program or an error message that indicates an invalid program.

The parse tree and derivation are equivalent for an unambiguous grammar.

It is useful to visualize the parser as building a parse tree. The parse tree's root is known; it represents the grammar's start symbol. The leaves of the parse tree are known; they match the stream of words returned by the scanner. The hard part of parsing lies in finding the connection between the leaves and the root. Two distinct and opposite approaches for constructing the tree suggest themselves:

1. *Top-Down Parsers* begin with the root and grow the tree toward the leaves. At each step, a top-down parser selects a node for some nonterminal on the lower fringe of the partially built tree and extends it with a subtree that represents the right-hand side of a production that rewrites the nonterminal.
2. *Bottom-Up Parsers* begin with the leaves and grow the tree toward the root. At each step, a bottom-up parser finds a substring of the partially built parse tree's upper fringe that matches the right-hand side of some production; it then builds a node for the rule's left-hand side and connects it into the tree.

In either scenario, the parser makes a series of choices about which productions to apply. Most of the intellectual complexity in parsing lies in the mechanisms for making these choices. Section 3.3 explores the top-down parsing, while Section 3.4 examines bottom-up parsing.

SECTION REVIEW

Language designers and compiler writers use context-free grammars to express programming-language syntax. These grammars allow specification of structure and form. They have shown themselves capable of expressing almost all of the programming language features that have proven useful.

This section defined context-free grammars and showed how they can be used to encode useful structure, such as blocking and precedence. It introduced ambiguity and the classic example of an ambiguity, the `if–then–else` grammar. Finally, it introduced the dual notions of derivations and parses—the subject of the rest of this chapter.

REVIEW QUESTIONS

1. Write a CFG to describe Backus-Naur Form.
2. Consider the classic expression grammar. What properties of the grammar and the input program govern the length of a derivation?

3.3 TOP-DOWN PARSING

A top-down parser begins with the root of the parse tree and systematically extends the tree downward until the leaves match the words returned by the scanner. At each point, the process considers the partially built parse tree. It picks a nonterminal symbol on the tree's lower fringe and extends the fringe with children that correspond to the right-hand side of some production for that nonterminal. It cannot extend the frontier from a terminal symbol. This process continues until either:

a. the fringe of the parse tree contains only terminal symbols, and the input stream has been exhausted, or

b. a clear mismatch occurs between the fringe of the partially built parse tree and the input stream.

In the first case, the parse succeeds. In the second case, two situations are possible. The parser may have selected the wrong production at some earlier step in the process, in which case it can backtrack, systematically reconsidering earlier decisions. For an input string that is a valid sentence, backtracking will eventually lead the parser to a correct sequence of choices and let it construct a correct parse tree. Alternatively, if the input string is not a valid sentence, backtracking will fail and the parser should report the syntax error to the user.

```
root ← node for the start symbol, S
focus ← root
push null onto the stack
word ← NextWord( )

while (true) do
    if (focus is a nonterminal) then
        pick next rule to expand focus, say $A \rightarrow \beta_1 \beta_2 \beta_3 \ldots \beta_n$
        build nodes for $\beta_1 \beta_2 \beta_3 \ldots \beta_n$ as children of focus
        push $\beta_n$, $\beta_{n\text{-}1}$, $\beta_{n\text{-}2}$, ... $\beta_2$ onto the stack
        focus ← $\beta_1$

    else if (word matches focus) then
        word ← NextWord( )
        focus ← pop from the stack

    else if (word = eof and focus = null)
        then accept the input and return root

    else backtrack
```

■ **FIGURE 3.3** A Leftmost, Top-Down Parsing Algorithm.

One key insight makes top-down parsing efficient: *a large subset of the context-free grammars can be parsed without backtracking.* Section 3.3.1 shows transformations that can often convert an arbitrary grammar into one suitable for backtrack-free top-down parsing. The subsequent sections introduce two kinds of top-down parsers: hand-coded recursive-descent parsers and generated LL(1) parsers.

The top-down parser builds a leftmost derivation to match the scanner's left-to-right scan of the input.

To make this discussion concrete, Fig. 3.3 sketches a high-level algorithm for a top-down parser that constructs a leftmost derivation. It builds a parse tree, anchored at the variable *root*. It uses a stack to track the unmatched portion of the lower fringe of the partially built parse tree.

The body of the parser consists of a while loop that examines the leftmost unmatched symbol on the partially built parse tree's lower fringe—stored in *focus*. If *focus* holds a nonterminal symbol, the parser expands the tree downward; it picks a production, builds the tree for the right-hand side, and assigns the leftmost symbol from this tree to *focus*. If *focus* is a terminal symbol, it compares *focus* against the next word in the input. A match moves *focus* to the next symbol on the fringe and gets the next word.

If *focus* is a terminal symbol, but it does not match the input, the parser must backtrack. The parser sets *focus* to its parent in the partially built parse tree, disconnects its children, and discards them. If an untried rule remains with *focus* on its left-hand side, the parser expands *focus* by that rule. It builds children for each symbol on the right-hand side, pushes those symbols onto the stack in right-to-left order, and sets *focus* to point at the first child. If no untried rule remains, the parser moves up another level and tries again. When it runs out of possibilities, it reports a syntax error and quits. Backtracking increases the asymptotic cost of parsing; in practice, it is an expensive way to find syntax errors.

To facilitate finding the "next" rule, the parser can store the rule number in a nonterminal's node when it expands that node.

When it backtracks, the parser must also rewind the input stream. Fortunately, the partial parse tree encodes enough information to make this action efficient. The parser must place each matched terminal in the discarded production back into the input stream, an action it can take as it disconnects them from the parse tree in a traversal of the discarded children.

3.3.1 Transforming a Grammar

The efficiency of a top-down parser depends critically on its ability to pick the correct production each time that it expands a nonterminal. If the parser always chooses correctly, top-down parsing is efficient. If it chooses poorly, the cost of parsing rises. This section examines structural issues that can make a CFGs unsuitable for a top-down parser. It presents transformations that the compiler writer can apply to the grammar to avoid these problems.

A Top-Down Parser with Oracular Choice

As an initial exercise, consider the behavior of the parser from Fig. 3.3 with the classic expression grammar in Fig. 3.2 when applied to the string `a + b × c`. For the moment, assume that the parser has an oracle that picks the correct production at each point in the parse. With oracular choice, it might proceed as shown in Fig. 3.4. The right column shows the input string, with a marker ↑ to indicate the parser's current position in the string. The symbol → in the rule column represents a step in which the parser matches a terminal symbol against the input string and advances the input. At each step, the sentential form represents the lower fringe of the partially built parse tree. For `a + b × c`, the parser applied eight rules and matched five words.

Notice, however, that oracular choice was inconsistent choice. In the first two steps, *focus* held the nonterminal *Expr*. In the first step, it applied rule 1, *Expr* → *Expr* + *Term*. The second step still had *Expr* as *focus* and `name` as the lookahead symbol. This time, it applied rule 3, *Expr* → *Term*. The oracle was similarly inconsistent in matching `names`, using the sequence (6,9) to match `a` from a *Term*, and then using the sequence (4,6,9) to match `b` from a *Term*. It would be difficult to make the top-down parser work well with consistent, algorithmic choice for this version of the expression grammar.

Lookahead symbol
the word that the parser is trying to match

In Fig. 3.4, the lookahead symbol is the word that immediately follows the ↑.

Rule	Sentential Form	Input
	Expr	↑ name + name × name
1	*Expr* + *Term*	↑ name + name × name
3	*Term* + *Term*	↑ name + name × name
6	*Factor* + *Term*	↑ name + name × name
9	name + *Term*	↑ name + name × name
→	name + *Term*	name ↑ + name × name
→	name + *Term*	name + ↑ name × name
4	name + *Term* × *Factor*	name + ↑ name × name
6	name + *Factor* × *Factor*	name + ↑ name × name
9	name + name × *Factor*	name + ↑ name × name
→	name + name × *Factor*	name + name ↑ × name
→	name + name × *Factor*	name + name × ↑ name
9	name + name × name	name + name × ↑ name
→	name + name × name	name + name × name ↑

■ **FIGURE 3.4** Leftmost, Top-Down Parse of a + b × c with Oracular Choice.

Eliminating Left Recursion

Left recursion

A CFG is *left recursive* if, for some $A \in NT$ and $\beta \in (T \cup NT)^*$, $A \rightarrow^+ A\beta$. That is, in one or more steps, the grammar can derive a sentential form from A that begins with A.

Direct left recursion occurs in one derivation step. Indirect left recursion takes more than one step.

The classic expression grammar has multiple problems that arise when used with a leftmost, top-down parser. Perhaps the most difficult problem arises from the fact that it employs *left recursion*. That is, it has productions, such as *Expr* → *Expr* + *Term*, whose right-hand sides begin with the nonterminal symbol from their left-hand sides.

Consider how the top-down parser would behave on the input a + b × c if it always applied productions in their order of appearance in the grammar. Its first several actions would be:

Rule	Sentential Form	Input
	Expr	↑ name + name × name
1	*Expr* + *Term*	↑ name + name × name
1	*Expr* + *Term* + *Term*	↑ name + name × name
1	···	↑ name + name × name

It starts with *Expr* and tries to match a. It applies rule 1 to create the sentential form *Expr* + *Term* on the fringe. Now, it faces the nonterminal *Expr* and the input word a, again. By consistent choice, it applies rule 1 to replace *Expr* with *Expr* + *Term*. Of course, it still faces *Expr* and the input word a. With this grammar and consistent choice, the parser will continue to expand the fringe indefinitely. The underlying problem is that the expansion by *Expr* + *Term* never generates a leading terminal symbol.

This problem arises from the use of left-recursion in productions 1 and 2. The same issue occurs in productions 4 and 5. With a left-recursive grammar, the top-down parser can expand the frontier indefinitely without generating a leading terminal symbol that the parser can either match or reject. To fix this problem, a compiler writer can convert the left-recursive grammar so that it uses only *right-recursion*.

Right recursion
A CFG is *right recursive* if, for some $A \in NT$ and $\beta \in (T \cup NT)^*$, $A \rightarrow^+ \beta A$.

The translation from left recursion to right recursion is mechanical. The example shown below illustrates the transformation with a direct left recursion. The left side contains a simple left-recursive grammar. The right side shows the grammar after transformation; the new grammar defines the same language as the original grammar.

$$\begin{array}{lcl} \textit{Fee} & \rightarrow & \textit{Fee}\ \alpha \\ & \mid & \beta \end{array} \qquad\qquad \begin{array}{lcl} \textit{Fee} & \rightarrow & \beta\ \textit{Fee}' \\ \textit{Fee}' & \rightarrow & \alpha\ \textit{Fee}' \\ & \mid & \epsilon \end{array}$$

The transformation introduces a new nonterminal, *Fee*′. It rewrites the first production to create a left-recursion on *Fee*′ rewrites the second production to generate *Fee*′ after β. It uses a production $\textit{Fee}' \rightarrow \epsilon$, where ϵ represents the empty string, to terminate the recursion. Such a production is called an *ϵ-production*.

In the classic expression grammar, direct left recursion appears in the productions for both *Expr* and *Term*.

Original Grammar			Transformed Grammar		
Expr	→	*Expr* + *Term*	*Expr*	→	*Term Expr*′
	|	*Expr* - *Term*	*Expr*′	→	+ *Term Expr*′
	|	*Term*		|	- *Term Expr*′
				|	ϵ
Term	→	*Term* × *Factor*	*Term*	→	*Factor Term*′
	|	*Term* ÷ *Factor*	*Term*′	→	× *Factor Term*′
	|	*Factor*		|	÷ *Factor Term*′
				|	ϵ

Plugging the transformed sequences back into the original grammar yields a right-recursive grammar that specifies the same language as the original grammar. Fig. 3.5 shows the right-recursive grammar.

Note that we did not simply rewrite *Expr* → *Expr* + *Term* as *Expr* → *Term* + *Expr*. That change would effectively change the associativity of addition (see Section 5.7.1).

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr′*
2	*Expr′*	→	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	→	*Factor Term′*
6	*Term′*	→	× *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

■ **FIGURE 3.5** Right-Recursive Variant of the Classic Expression Grammar.

The transformation eliminates direct left recursion. Left recursion can also occur indirectly as $\alpha \rightarrow^{+} \alpha\delta$, through a chain of rules such as $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$, and $\gamma \rightarrow \alpha\delta$. Indirect left recursion is not always obvious; it can be obscured by a long chain of productions.

To eliminate all left recursion, we need a systematic approach. The algorithm in Fig. 3.6 eliminates all left recursion from a grammar by thorough application of two techniques: forward substitution to convert indirect left recursion into direct left recursion and conversion of direct left recursion to right recursion with the transformation, as before. The algorithm, as stated, assumes that the original grammar has no cycles ($A \rightarrow^{+} A$) and no ϵ-productions.

The algorithm imposes an arbitrary order on the nonterminals in the grammar. The outer loop cycles through the nonterminals in that order. The inner loop looks for any production that expands A_i into a right-hand side that begins with A_s, for $s < i$. Such an expansion may lead to an indirect left recursion. To avoid this, the algorithm replaces the occurrence of A_s with all the alternative right-hand sides for A_s. This process eventually converts each indirect left recursion into a direct left recursion. The final step in the outer loop converts any direct left recursion on A_i to right recursion using the simple transformation shown earlier. Because new nonterminals are added at the end and only involve right recursion, the loop can ignore them—they do not need to be checked and converted.

Considering the loop invariant for the outer loop may make this clearer. At the start of the ith outer loop iteration

$$\forall\ k < i, \text{ no production expanding } A_k \text{ has } A_s \text{ in its } rhs, \text{ for } s < k.$$

At the end of this process, ($i = n$), all indirect left recursion has been eliminated through forward substitution, and all direct left recursion has been eliminated with the transformation for direct left recursion.

```
impose an order on the nonterminals, $A_0, A_1, \ldots, A_i$
for $i \leftarrow 0$ to $n$ do
    for $s \leftarrow 0$ to $i$-1 do
        replace each production $A_i \rightarrow A_s\,\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
            where $A_i \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are the current productions for $A_i$
    eliminate any direct left recursion on $A_i$ using the transformation
```

■ **FIGURE 3.6** Algorithm for Removal of Indirect Left Recursion.

Example

Consider the simple left-recursive grammar shown in the margin. Subscripts on the nonterminal symbols indicate the order used by the algorithm. The algorithm proceeds as follows:

0	$Goal_0$	→	A_1
1	A_1	→	B_2 a
2		|	a
3	B_2	→	A_1 b

Original Grammar

0	$Goal_0$	→	A_1
1	A_1	→	B_2 a
2		|	a
3	B_2	→	a b C_3
4	C_3	→	a b C_3
5		|	ϵ

Transformed Grammar

$i = 0$ The algorithm does not enter the inner loop.
No rule of the form $Goal_0 \rightarrow Goal_1\,\gamma$ found.

$i = 1$
$s = 0$ The algorithm looks for a rule with the form $A_1 \rightarrow Goal_0\,\gamma$. No such rule exists.

$i = 2$
$s = 0$ The algorithm looks for a rule with the form $B_2 \rightarrow Goal_0\,\gamma$. No such rule exists.

$i = 2$
$s = 1$ The algorithm looks for a rule with the form $B_2 \rightarrow A_1\,\gamma$. It finds rule 3 and rewrites it as $B_2 \rightarrow B_2$ a b | a b.

It then applies the transformation for direct left recursion to these new rules. This process produces rules 3, 4, and 5 in the final grammar. It introduces the new symbol C_3.

Parsing with Epsilon Productions

The technique for eliminating left recursion introduced ϵ-productions. The top-down parser, from Fig. 3.3, does not understand ϵ-productions. Fortunately, the extension to handle them is minor. When the parser picks a production whose right-hand side is ϵ, it should simply set $focus \leftarrow pop()$. This simple action advances its attention to the next node, terminal or nonterminal, on the fringe.

Backtrack-Free Parsing

The major source of inefficiency in the top-down parser shown in Fig. 3.3 is that it may need to backtrack. As written, it has no mechanism to pick the correct production by which to expand the lower fringe of the partially built parse tree. (The algorithm simply says "*pick the next rule* ...".) A correct pick leads to an efficient parse. An incorrect pick leads to a mismatch between the fringe and the input, followed by a cycle of retracting and reexpanding the fringe until it finds the right pick or discovers, exhaustively, that no correct pick exists.

To make top-down parsing efficient, the algorithm needs a mechanism that determines, in one step, either the correct expansion or the fact that no correct expansion exists. It must, in some sense, implement oracular choice.

9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

Rules for Factor

Consider the rules to expand *Factor* in the expression grammar, repeated in the margin. The top-down parser, trying to expand *Factor* to match a name must pick rule 11. Any consistent ordering among rules 9, 10, and 11 will produce the wrong choice (and backtrack) on at least two of the symbols: (, num, or name. The solution seems obvious: the parser should use the next input symbol—the lookahead symbol—to inform its choice. In this case, the combination of the current *focus* and the lookahead symbol let it pick the correct production.

We can modify the top-down parser from Fig. 3.3 so that it always picks the correct production in the grammar for *Factor*. Does this mechanism generalize to other grammars?

> *The top-down parser can pick the correct production if, at every step in the derivation, the grammar has the property that the lookahead symbol uniquely specifies the correct choice.*

Backtrack-free grammar
a CFG for which a leftmost, top-down parser can always predict the correct rule with bounded lookahead

A backtrack-free grammar is sometimes called a *predictive grammar*.

For such a grammar, the top-down parser can disambiguate all of the choices with the combination of the current *focus* and a one-symbol lookahead. We call such a grammar *backtrack free* and note that it requires a one-symbol lookahead. A large subset of the CFGs are backtrack free.

To make efficient deterministic choices, the parser must know, for each possible expansion of a nonterminal A, the set of terminal symbols that can occur as the first symbol in a string derived from that expansion. If $A \rightarrow \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_n$, the parser needs to know which terminal symbols can occur at the start of a sentential form derived from each of the γ_i. If, for each pair of right-hand sides, the sets of such symbols are disjoint, then the top-down parser can always choose the correct expansion for A.

The intuition is clear. In the grammar for *Factor*, each right-hand side starts with a unique terminal symbol, so the disambiguating sets each contain one terminal. If, however, some right-hand side starts with a nonterminal, the construction of those sets is more difficult.

```
for each $\alpha \in (T \cup$ eof $\cup \epsilon)$ do
    FIRST($\alpha$) $\leftarrow \alpha$

for each $A \in NT$ do
    FIRST($A$) $\leftarrow \emptyset$

while (FIRST sets are still changing) do

    for each $p \in P$, of the form $A \rightarrow \beta_1 \beta_2 \ldots \beta_k$ do
        rhs $\leftarrow$ FIRST($\beta_1$) $- \{\epsilon\}$
        trailing $\leftarrow$ true

        for i $\leftarrow$ 1 to k-1 do
            if $\epsilon \in$ FIRST($\beta_i$)
                then rhs $\leftarrow$ rhs $\cup$ (FIRST($\beta_{i+1}$) $- \{\epsilon\}$)
                else
                    trailing $\leftarrow$ false
                    break

        if trailing and $\epsilon \in$ FIRST($\beta_k$) then
            rhs $\leftarrow$ rhs $\cup \{\epsilon\}$

        FIRST($A$) $\leftarrow$ FIRST($A$) $\cup$ rhs
```

■ **FIGURE 3.7** Computing FIRST Sets for Symbols in a Grammar.

For each grammar symbol α, define the set FIRST(α) as the set of terminal symbols that can appear as the first word in some sentential form derived from α. The domain of FIRST is $\{NT \cup T \cup \epsilon \cup$ eof$\}$, the set of all grammar symbols. Its range is $\{T \cup \epsilon \cup$ eof$\}$, the set of grammar symbols, minus the nonterminal symbols. (Note that $\{T \cup$ eof$\}$ is the set of words that the scanner returns.) If $\alpha \in \{T \cup \epsilon \cup$ eof$\}$, then FIRST(α) $= \{\alpha\}$. For a nonterminal A, FIRST(A) contains the complete set of terminal symbols that can appear as the leading symbol in a sentential form derived from A.

FIRST set
For a grammar symbol α, FIRST(α) is the set of terminals that can appear at the left end of a sentential form derived from α.

We defined FIRST sets over single grammar symbols. It is convenient to extend that definition of FIRST from a single grammar symbol to a string of symbols. For a string of symbols, $s = \beta_1 \beta_2 \beta_3 \ldots \beta_k$, we define FIRST($s$) as the union of the FIRST sets for $\beta_1 \beta_2 \beta_3 \ldots \beta_i$, where β_i is the first symbol whose FIRST set does not contain ϵ. Further, $\epsilon \in$ FIRST(s) if and only if it is in FIRST(β_i) for each β_i, $1 \leq i \leq k$.

Fig. 3.7 shows an algorithm to compute FIRST sets. As its initial step, the algorithm fills in the FIRST sets for the terminal symbols, ϵ, and eof. For the right-recursive expression grammar shown in Fig. 3.5, that initial step produces the following FIRST sets:

eof implicitly ends every sentence. Thus, it is in both the domain and range of FIRST.

	num	**name**	+	-	×	÷	(	)	**eof**	ϵ
FIRST	num	name	+	-	×	÷	(	)	eof	ϵ

Next, the algorithm iterates over the productions. For each production, $A \rightarrow \beta$, it computes FIRST(β), using the extension from single symbol to string of symbols. The algorithm computes the FIRST set of the entire right-hand side into *rhs* and then adds *rhs* to the FIRST set of the nonterminal symbol on the production's left-hand side. This process halts when it reaches a fixed point. For the right-recursive expression grammar, the FIRST sets of the nonterminals are:

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	×, ÷ , ϵ	(, name, num

Conceptually, FIRST sets simplify implementation of a top-down parser. In the subgrammar for *Factor*, for example, the FIRST sets make obvious the correct choice among the productions.

		Productions		**FIRST Set**
9	*Factor*	→	(*Expr*)	{ (}
10		|	num	{ num }
11		|	name	{ name }

For the *Term′* subgrammar, however, the choice is less clear.

		Productions		**FIRST Set**
6	*Term′*	→	× *Factor* *Term′*	{ × }
7		|	÷ *Factor* *Term′*	{ ÷ }
8		|	ϵ	{ ϵ }

For rules 6 and 7, the choices are clear. Rule 8 poses a harder problem. FIRST(ϵ) does not match any word returned by the scanner.

Intuitively, the parser should apply the ϵ production when the lookahead symbol is not a member of the FIRST set of any of the other alternative productions—in this case, when the lookahead symbol is neither "×" nor "÷". However, the parser should not accept just any word; some words will lead to derivations while others lead to syntax errors. To differentiate between these cases, the set used for rule 8 should contain any word that can

```
for each $A \in NT$ do
    FOLLOW($A$) $\leftarrow \emptyset$

FOLLOW($S$) $\leftarrow$ { eof }

while (FOLLOW sets are still changing) do
    for each $p \in P$ of the form $A \to \beta_1 \beta_2 \beta_3 \ldots \beta_k$ do
        TRAILER $\leftarrow$ FOLLOW($A$)
        for $i \leftarrow k$ down to 1 do
            if $\beta_i \in NT$ then
                FOLLOW($\beta_i$) $\leftarrow$ FOLLOW($\beta_i$) $\cup$ TRAILER
                if $\epsilon \in$ FIRST($\beta_i$) then
                    TRAILER $\leftarrow$ TRAILER $\cup$ (FIRST($\beta_i$) $- \epsilon$)
                else TRAILER $\leftarrow$ FIRST($\beta_i$)
            else TRAILER $\leftarrow$ { $\beta_i$ }   // $\beta_i \in T$
```

■ **FIGURE 3.8** Computing FOLLOW Sets for Nonterminal Symbols.

appear as the leading symbol after a valid application of rule 8—the set of words that can follow a *Term'*.

To capture that knowledge of the grammar's structure, we define the set FOLLOW(A) to contain all of the words that can occur to the immediate right of a string derived from A. The domain of FOLLOW is NT and its range is $T \cup$ {eof}. Fig. 3.8 presents an algorithm that computes the FOLLOW sets for a grammar; it uses the FIRST sets.

FOLLOW set
For a nonterminal α, FOLLOW(α) contains the set of words that can occur immediately after α in a sentence.

The FOLLOW set algorithm is more subtle than the FIRST set algorithm. To begin, it sets each FOLLOW set to the empty set. It then sets the start symbol's FOLLOW set to { eof }. The main part of the algorithm iterates over the individual productions, refining FOLLOW sets. It halts when those sets reach a fixed point.

The construction uses adjacency in the right-hand side to update the FOLLOW sets of nonterminals in that right-hand side. If $\beta_i \, \beta_{i+1}$ appears in some right-hand side, then FOLLOW(β_i) should contain FIRST(β_{i+1}). Rules with ϵ complicate matters. If $\epsilon \in$ FIRST(β_{i+1}), then FOLLOW(β_i) should also contain the symbols that can follow β_{i+1}.

To capture this effect, the algorithm iterates over the right-hand side of each production, $A \to \beta_1 \beta_2 \ldots \beta_k$ from β_k to β_1. For each suffix,

$$\beta_k \, ; \, \beta_{k\text{-}1} \, \beta_k \, ; \, \ldots ; \, \beta_1 \, \beta_2 \ldots \beta_k$$

it constructs the FIRST set. The variable *TRAILER* holds those FIRST sets; if some β_i can derive ϵ, *TRAILER* carries forward the right context. If β_i

Production			START Set
Goal	→	*Expr*	{ (, name, num }
Expr	→	*Term Expr′*	{ (, name, num }
Expr′	→	+ *Term Expr′*	{ + }
	|	- *Term Expr′*	{ - }
	|	ϵ	{ eof,) }
Term	→	*Factor Term′*	{ (, name, num }
Term′	→	× *Factor Term′*	{ × }
	|	÷ *Factor Term′*	{ ÷ }
	|	ϵ	{ eof, +, -,) }
Factor	→	(*Expr*)	{ (}
	|	num	{ num }
	|	name	{ name }

■ **FIGURE 3.9** START Sets for the Right-Recursive Expression Grammar.

cannot derive ϵ, the algorithm truncates that right context and sets *TRAILER* to FIRST(β_i).

The FOLLOW sets for the right-recursive expression grammar are:

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, ×, ÷,)

We introduced FOLLOW sets to allow the parser to make correct expansions of ϵ productions, such as the subgrammar for *Term′*. The parser must choose between three right-hand sides for *Term′*.

FIRST(× *Factor Term′*) = { × }
FIRST(÷ *Factor Term′*) = { ÷ }

The FIRST sets make the choice clear for a lookahead of either × or ÷. The difficult choice arises when the lookahead is neither × nor ÷. Does the parser expand by ϵ or does it throw a syntax error?

FOLLOW sets let the parser make this decision. If the lookahead symbol $l \in$ FOLLOW(*Term′*), the parser should expand by ϵ. If l is not × or ÷ and $l \notin$ FOLLOW(*Term′*), then the input contains a syntax error.

The combination of FIRST and FOLLOW let us define precisely the correct behavior for the parser. For some production $A \rightarrow \beta$:

$$\text{START}(A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta), & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ (\text{FIRST}(\beta) - \epsilon) \cup \text{FOLLOW}(A), & \textit{otherwise.} \end{cases}$$

Fig. 3.9 shows the START sets for the right-recursive expression grammar. A top-down parser can use START sets to pick the correct expansion for any nonterminal *A* by comparing the lookahead symbol to the START sets of the alternative right-hand sides for *A*.

This observation leads to a simple and effective test to determine whether or not a grammar can be parsed, left-to-right, with a top-down parser without the need to backtrack. A grammar is *backtrack free* if and only if, for any nonterminal *A* with multiple right-hand sides, $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$:

$$\text{START}(A \rightarrow \beta_i) \cap \text{START}(A \rightarrow \beta_j) = \emptyset, \quad \forall\ 1 \le i, j \le n, \quad i \ne j.$$

If a grammar has this property, a top-down parser can alway choose the correct expansion for a nonterminal *A* with a single lookahead symbol. This property is known as the LL(1) condition, because grammars that meet this condition can be parsed with an LL(1) parser (see Section 3.3.3).

Left-Factoring to Eliminate Backtracking

Not all grammars are backtrack free. For example, consider extending the expression grammar to include function calls, denoted with (and), and array-element references, denoted with [and]. We can expand *Factor* → `name` into three rules, plus appropriate productions to describe the argument lists.

Factor → `name` is rule 11 in Fig. 3.5.

11	*Factor*	→	`name`
12		|	`name` [*ArgList*]
13		|	`name` (*ArgList*)
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	, *Expr MoreArgs*
17		|	ϵ

Because rules 11, 12, and 13 all begin with `name`, their START sets are identical. When the parser tries to expand an instance of *Factor* with a lookahead of `name`, it has no basis to choose among 11, 12, and 13. Single-word lookahead is not enough to resolve the choice. In this case, we can rewrite the grammar to create disjoint START sets.

A two-word lookahead would handle this case. However, for any finite lookahead, we can devise a grammar where that lookahead is insufficient.

11	*Factor*	→	`name` *Arguments*
12	*Arguments*	→	[*ArgList*]
13		|	(*ArgList*)
14		|	ϵ

This rewrite, called *left factoring*, breaks the derivation of *Factor* into two steps. The first step matches the common prefix of the original rules 11, 12,

Left factoring
the process of extracting and isolating common prefixes in a set of productions

and 13. The second step recognizes the three distinct suffixes: [*ArgList*], (*ArgList*), and ϵ. It adds a nonterminal, *Arguments*, and pushes the alternate suffixes for *Factor* into expansions of *Arguments*.

If a nonterminal symbol has multiple right-hand sides that share a common prefix, we can left factor the rules to shift recognition of the common prefix onto a new nonterminal symbol. The transformation takes a nonterminal and its productions:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j$$

where α is the common prefix and the various γ_i's represent right-hand sides that do not begin with α. The transformation introduces a new nonterminal B to represent the alternate suffixes for α and rewrites the original productions according to the pattern:

$$\begin{aligned} A &\rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j \\ B &\rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \end{aligned}$$

To left factor a complete grammar, we must systematically inspect each nonterminal, discover common prefixes, and apply the transformation. For example, in the pattern above, we must consider factoring the right-hand sides of B, as two or more of the β_i's could share a prefix. The process stops when all common prefixes have been removed.

Left-factoring can often eliminate the need to backtrack. However, some context-free languages have no backtrack-free grammar. They must be parsed with a more general technique, such as the LR(1) parsers described in the next section. An example of a language that has no backtrack-free grammar but can be parsed by an LR(1) parser is given in Exercise 3.10.

Given an arbitrary CFG, the compiler writer can systematically eliminate left recursion and left-factor common prefixes. These changes may produce a backtrack-free grammar. In general, however, it is undecidable whether or not a backtrack-free grammar exists for an arbitrary context-free language.

3.3.2 Top-Down Recursive-Descent Parsers

Backtrack-free grammars lend themselves to simple and efficient parsing with a paradigm called *recursive descent*. A recursive-descent parser is structured as a set of mutually recursive procedures, one for each nonterminal in the grammar. The procedure that corresponds to nonterminal A recognizes an instance of A in the input stream. To recognize a nonterminal B on some right-hand side for A, the parser invokes the procedure that corresponds to B. Thus, the grammar itself guides the parser's implementation.

```
Main() /* Goal → Expr */
    word ← NextWord();
    if (Expr())
        then if (word = eof)
            then report success;
            else Fail();

Fail()
    report syntax error;
    attempt error recovery or exit;

Expr() /* Expr → Term Expr′ */
    if (Term())
        then return ExprPrime();
        else Fail();

ExprPrime()
    /* Expr′ → + Term Expr′ | - Term Expr′ */
    if (word = + or word = -) then
        word ← NextWord();
        if (Term())
            then return ExprPrime();
            else Fail();
    /* Expr′ → ϵ */
    else if (word = ) or word = eof)
        then return true;
        else Fail();

Term() /* Term → Factor Term′ */
    if (Factor())
        then return TermPrime();
        else Fail();
```

```
TermPrime()
    /* Term′ → × Factor Term′ | ÷ Factor Term′ */
    if (word = × or word = ÷) then
        word ← NextWord();
        if (Factor())
            then return TermPrime();
            else Fail();
    /* Term′ → ϵ */
    else if (word = + or word = - or
             word = ) or word = eof)
        then return true;
        else Fail();

Factor()
    /* Factor → ( Expr ) */
    if (word = ( ) then
        word ← NextWord();
        if (not Expr())
            then Fail();
        if (word ≠ ) )
            then Fail();
        word ← NextWord();
        return true;
    /* Factor → num | name */
    else if (word = num or
            word = name) then
            word ← NextWord();
            return true;
    else Fail();
```

■ **FIGURE 3.10** Recursive-Descent Parser for Expressions.

Recall the rules for *Term′* in the expression grammar:

		Production		START Set
6	*Term′*	→	× *Factor Term′*	{ × }
7		|	÷ *Factor Term′*	{ ÷ }
8		|	ϵ	{ eof, +, -,) }

To recognize instances of *Term′*, we will create a routine *TermPrime()*. It follows a simple scheme: choose among the three rules (or a syntax error)

```
word ← NextWord()
push eof onto Stack
push the start symbol, S, onto Stack

while( true ) do
    focus ← top of Stack

    if (focus = eof and word = eof)
        then report success and break from the loop

    else if (focus ∈ T or focus = eof) then
        if focus matches word then
            pop Stack
            word ← NextWord()

        else report an error looking for the symbol in focus

    else    // focus is a nonterminal
        if Table[focus, word] is $A \rightarrow \beta_1 \beta_2 \ldots \beta_k$ then
            pop Stack
            for i ← k to 1 by -1 do
                if ($\beta_i \neq \epsilon$) then
                    push $\beta_i$ onto Stack

        else report an error expanding focus
```

■ **FIGURE 3.11** The Skeleton LL(1) Parser.

based on the START sets of their right-hand sides. For each right-hand side, the code tests directly for any further symbols.

3.3.3 Table-Driven LL(1) Parsers

To test for the presence of a nonterminal, say *A*, the code invokes the procedure that corresponds to *A*. To test for a terminal symbol, such as name, it performs a direct comparison and, if successful, advances the input stream by calling the scanner, *NextWord()*. If it matches an ϵ-production, the code does not call *NextWord()*. Fig. 3.10 shows pseudocode for a recursive descent parser for expressions. Look at the implementation of *TermPrime* in the upper right corner; it follows this scheme with explicit tests against the symbols in the various START sets. It combines the code for rules 6 and 7.

The strategy for constructing a complete recursive-descent parser is clear. For each nonterminal, we construct a procedure to recognize its alternative right-hand sides. These procedures call one another to recognize nonterminals. They recognize terminals by direct matching. Each routine returns an indicator of success or it calls *Fail*.

	`eof`	+	-	×	÷	(	)	`num`	`name`
Goal						0		0	0
Expr						1		1	1
Expr′	4	2	3				4		
Term						5		5	5
Term′	8	8	8	6	7		8		
Factor						9		10	11

■ **FIGURE 3.12** LL(1) Parse Table for the Right-Recursive Expression Grammar.

For a small grammar, a compiler writer can quickly craft a recursive-descent parser. A recursive-descent parser can produce accurate, informative error messages. The natural location for generating those messages is when the parser fails to find an expected terminal symbol—inside *ExprPrime*, *TermPrime*, and *Factor* in the example.

Given START sets, we can automate generation of top-down parsers for backtrack-free grammars. To do so, we build a tool, a *parser generator*, that constructs FIRST, FOLLOW, and START sets, and then uses the START sets to construct a top-down parser.

Parser generator
a tool that builds a parser from specifications, usually a BNF-like grammar

One scheme to build top-down parsers creates table-driven LL(1) parsers. Any backtrack-free grammar is usable in an LL(1) parser. The name LL(1) derives from the fact that these parsers scan their input Left to right, discover a Leftmost derivation, and use a 1 symbol lookahead.

Backtrack-free grammars are often called LL(1) grammars.

To build an LL(1) parser, the compiler writer creates a right-recursive, backtrack-free grammar. A *parser generator* constructs the actual parser. Most LL(1) parser generators use a table-driven skeleton parser, such as the one shown in Fig. 3.11. The parser generator constructs the table. Fig. 3.12 shows the LL(1) table for the right-recursive expression grammar.

In the skeleton parser, the variable `focus` holds the next grammar symbol that must be matched on the partially built parse tree's lower fringe—the same role it played in the top-down parser from Fig. 3.3. The LL(1) table maps pairs of nonterminals and lookahead symbols (terminals or `eof`) into productions. Given a nonterminal *A* and a lookahead symbol *w*, *Table[A, w]* specifies the correct expansion.

The example in Fig. 3.13(a) shows the actions of the LL(1) expression parser for the input string `a + b × c`. The central column shows the contents of the parser's stack, which holds the partially built lower fringe of the parse tree. The parse concludes successfully when it pops *Expr′* from the stack, leaving `eof` exposed on the stack and `eof`, implicitly, as the lookahead symbol.

Rule	Stack	Input
—	eof *Goal*	↑ name + name × name
0	eof *Expr*	↑ name + name × name
1	eof *Expr′* *Term*	↑ name + name × name
5	eof *Expr′* *Term′* *Factor*	↑ name + name × name
11	eof *Expr′* *Term′* name	↑ name + name × name
→	eof *Expr′* *Term′*	name ↑ + name × name
8	eof *Expr′*	name ↑ + name × name
2	eof *Expr′* *Term* +	name ↑ + name × name
→	eof *Expr′* *Term*	name + ↑ name × name
5	eof *Expr′* *Term′* *Factor*	name + ↑ name × name
11	eof *Expr′* *Term′* name	name + ↑ name × name
→	eof *Expr′* *Term′*	name + name ↑ × name
6	eof *Expr′* *Term′* *Factor* ×	name + name ↑ × name
→	eof *Expr′* *Term′* *Factor*	name + name × ↑ name
11	eof *Expr′* *Term′* name	name + name × ↑ name
→	eof *Expr′* *Term′*	name + name × name ↑
8	eof *Expr′*	name + name × name ↑
4	eof	name + name × name ↑

(a) Actions of the LL(1) Parser on a + b × c

Rule	Stack	Input
—	eof *Goal*	↑ name + ÷ name
0	eof *Expr*	↑ name + ÷ name
1	eof *Expr′* *Term*	↑ name + ÷ name
5	eof *Expr′* *Term′* *Factor*	↑ name + ÷ name
11	eof *Expr′* *Term′* name	↑ name + ÷ name
→	eof *Expr′* *Term′*	name ↑ + ÷ name
8	eof *Expr′*	name ↑ + ÷ name
2	eof *Expr′* *Term* +	name ↑ + ÷ name
→	eof *Expr′* *Term*	name + ↑ ÷ name

syntax error
at this point

(b) Actions of the LL(1) Parser on x + ÷ y

■ **FIGURE 3.13** Example LL(1) Parses.

```
for each nonterminal A do
    Table[A,eof] ← error
    for each terminal w do
        Table[A,w] ← error

    for each production p of the form A → β do
        for each terminal w ∈ START(A → β) do
            Table[A,w] ← p

        if eof ∈ START(A → β)
            then Table[A,eof] ← p
```

■ **FIGURE 3.14** LL(1) Table-Construction Algorithm.

As Fig. 3.13(a) shows, the LL(1) parser is efficient. It takes time proportional to the size of the derivation. It shifts every grammar symbol in the derivation onto the stack and later pops each of them off the stack. We cannot expect a parser to construct a derivation in fewer steps.

Now consider the actions of the LL(1) parser on the illegal input string x + ÷ y, shown in Fig. 3.13(b). It recognizes the first name and the +. At that point, it has *Term* at the top of the stack and a lookahead symbol of ÷. The corresponding table entry contains *error*.

Table has a row for each nonterminal symbol and a column for each terminal symbol. The algorithm to fill the table is straightforward. It assumes that START sets are available for the grammar.

The construction. shown in Fig. 3.14, assigns each production an ordinal number and initializes each table entry to *error*. Next, for each production $A \rightarrow \beta$ and each symbol $w \in$ START($A \rightarrow \beta$), it sets the table entry for row *A* and column *w* to contain the production's number. For the right-recursive expression grammar, this produces the table shown in Fig. 3.12.

For a grammar that meets the LL(1) condition (see page 111), this construction will produce a correct table in $\mathbf{O}(|P| \times |T|)$ time, where *P* is the set of productions and *T* is the set of terminals.

If the construction assigns multiple production numbers to *Table[A, w]*, then multiple right-hand sides for *A* have *w* in their START sets, violating the LL(1) condition. Since *Table* is initialized to *error*, the table builder can test before each assignment and report a problem if the entry is not set to *error*.

As an alternative to the table-driven parser, an LL(1) parser generator could simply emit a recursive-descent parser. The process would be similar to that used to construct a table-driven LL(1) parser. First, the parser generator would build FIRST, FOLLOW, and START sets. Next, it would iterate over the nonterminal symbols from the left-hand sides of the grammar rules. For each such nonterminal, it would emit a small procedure to recognize its various right-hand sides. The code would use the START sets to make decisions. Such a system could combine the speed and locality of a recursive-descent parser with the convenience of a grammar-based generator.

SECTION REVIEW

Top-down parsers are simple, compact, and efficient. They can be implemented in a number of ways, including hand-coded, recursive-descent parsers and generated LL(1) parsers. Because these parsers know, at each point in the parse, the set of words that can occur as the next symbol in a valid input string, they can produce accurate and useful error messages.

Most programming-language constructs can be expressed in a backtrack-free grammar. Thus, these techniques have widespread application. The restriction that alternate right-hand sides for a nonterminal have disjoint START sets does not seriously limit the utility of LL(1) grammars. The primary drawback of top-down, predictive parsers lies in their inability to handle left recursion.

REVIEW QUESTIONS

1. To build an efficient top-down parser, the compiler writer must express the source language in a somewhat constrained form. Explain the restrictions on the source-language grammar that are required to make it amenable to efficient top-down parsing.
2. Name two potential advantages of a hand-coded recursive-descent parser over a generated, table-driven LL(1) parser, and two advantages of the LL(1) parser over a recursive-descent implementation.

3.4 BOTTOM-UP PARSING

Bottom-up parsers discover a derivation by working from the words in the program toward the start symbol, S. We can think of this process as building the parse tree starting from its leaves and working toward its root. For a derivation:

$$\mathit{Start} = \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_{n\text{-}1} \rightarrow \gamma_n = \texttt{sentence}$$

a bottom-up parser repeatedly discovers, from γ_i and the grammar, the production $A \rightarrow \beta$ and the location k in γ_i such that replacing the occurrence of β that ends at position k in γ_i with A produces $\gamma_{i\text{-}1}$.

```
push INVALID on to the stack
word ← NextWord()
repeat until (top of stack = S and word = eof) do
    if the top of stack is a handle A → β then
        // reduce β to A
        pop |β| symbols off the stack
        push A onto the stack
    else if (word ≠ eof) then
        // shift word onto the stack
        push word onto the stack
        word ← NextWord()
    else  // parser needs to shift, but is out of input
        throw a syntax error
report success
```

■ **FIGURE 3.15** A Simple Shift-Reduce Parser.

The pair $\langle A \rightarrow \beta, k\rangle$ describes this transition from γ_i to $\gamma_{i\text{-}1}$. We call this pair a *handle*. The parser repeatedly finds a handle in the frontier and rewrites β with A. In a valid derivation from an unambiguous grammar, each γ_i should have exactly one handle, for all $0 < i \leq n$.

Handle
A handle of a sentential form γ_i is a pair, $\langle A \rightarrow \beta, k\rangle$, such that β appears in the sentential form with its right end at position k and replacing β with A produces the prior step in the derivation, $\gamma_{i\text{-}1}$.

A bottom-up parser operates by finding the series of handles that define a legal derivation for the input program. Fig. 3.15 shows a high-level algorithm for a shift-reduce parser.

The parser repeats a simple process. It repeatedly shifts words—⟨*lexeme*, *category*⟩ pairs from the scanner—onto a stack until it finds a handle on the stack. The handle will appear with its right-end at the top of the stack. When it finds a handle for $A \rightarrow \beta$, it pops the stack entries for β from the stack and pushes an entry for A onto the stack—a *reduce* action in the terminology of bottom-up parsing.

The parser continues shifting and reducing until either:

1. The parser finds the grammar's start symbol, S, on top of the stack. If the parser finds S on top of the stack and the lookahead word is `eof`, then the parse succeeds. If the lookahead word is not `eof`, then more input remains to be processed.
2. The scanner returns `eof` and the parser does not reduce the stack to S. In this case, it has consumed all of the input without finding a legal derivation. The parser should report a *syntax error*—that is, the input program contains one or more errors.

This error-finding mechanism does not localize the error. LR(1) parsers detect errors where they occur.

The key to practical bottom-up parsing is the ability to recognize handles efficiently. The LR(1) parsers implement a particularly efficient handle-finding mechanism. We will return to the subject of handles and handle-finding throughout Section 3.4. First, however, we will finish our high-level description of bottom-up parsers.

Consider an extension of the shift-reduce parser that builds an actual parse tree. Each shift action constructs a leaf node to represent the lookahead symbol, then advances the lookahead symbol by calling the scanner. Each reduce action adds a node that represents a nonterminal on top of the partially built parse tree. These interior nodes for nonterminals record the grammatical structure discovered in the derivation of the input string.

At any stage in the parse, the partially built parse tree represents the state of the parse. Each leaf node represents a word; reading the leaves left-to-right shows a prefix of the input program. Interior nodes, above the leaves, encode those parts of the derivation that the parser has found. The parser works along the upper frontier of this partially built parse tree; that frontier corresponds to the current sentential form in the derivation being built by the parser.

To extend the frontier upward, the parser finds a handle, $\langle A \rightarrow \beta,k\rangle$. Thus, β occurs on the frontier with its right end at position k, and the parser knows that replacing β with A will create, along the frontier, the previous sentential form in the derivation. (If the frontier represents γ_i, then the reduced frontier will represent γ_{i-1}.) This process constructs the derivation in reverse—that is, from the sentence to the start symbol rather than from the start symbol to the sentence.

Our conceptual parser builds a rightmost derivation, in reverse order. This order reconciles the scanner's left-to-right scan with the parser's rightmost derivation. In a rightmost derivation, the leftmost leaf is considered last. Reversing that order leads to the desired behavior: leftmost leaf first and rightmost leaf last.

At each point, the shift-reduce parser operates on the frontier of the partially built parse tree; the contents of the stack, read bottom-to-top, are a prefix of the corresponding sentential form in the derivation. Because each sentential form occurs in a rightmost derivation, the unexamined suffix—that part beyond the top of the stack—consists entirely of terminal symbols. When the parser needs more right context, it calls the scanner.

With an unambiguous grammar, the rightmost derivation is unique. For a large class of unambiguous grammars, γ_{i-1} can be determined directly from γ_i (the parse tree's upper frontier) and a limited lookahead into the input

```
push ⟨INVALID, INVALID⟩ onto the stack
push ⟨start symbol, $s_0$⟩ onto the stack
word ← NextWord( )

while (true) do
    state ← state from pair at top of stack

    if Action[state,word] = "reduce $A \to \beta$" then
        pop $|\beta|$ pairs from the stack
        state ← state from pair at top of stack
        push ⟨A, Goto[state, A]⟩ onto the stack

    else if Action[state,word] = "shift $s_i$" then
        push ⟨word, $s_i$⟩ onto the stack
        word ← NextWord( )

    else if Action[state,word] = "accept" and word = eof
        then break

    else throw a syntax error

report success /* executed the "accept" case */
```

■ **FIGURE 3.16** The Skeleton LR(1) Parser.

stream. In other words, given a frontier γ_i and a limited amount of lookahead, the parser can find, efficiently and deterministically, the handle that takes γ_i to $\gamma_{i\text{-}1}$.

LR(1) parsers are a particularly efficient implementation of these handle-finding, shift-reduce parsers. An LR(1) parser scans the input from left to right to build a rightmost derivation in reverse. At each step, it makes decisions based on the history of the parse and a lookahead of, at most, one symbol. The name LR(1) derives from these properties: Left-to-right scan, Reverse rightmost derivation, and 1 symbol of lookahead. Tools that build LR(1) parsers are widely available.

Informally, we will say that a language has the LR(1) property if it can be parsed in a single left-to-right scan to build a reverse-rightmost derivation using just one symbol of lookahead to determine parsing actions. In practice, the simplest test to determine if a grammar has the LR(1) property is to let a parser generator try to build the LR(1) parser. If it fails, the grammar is not an LR(1) grammar.

The remainder of this section presents a detailed introduction to the theory of LR(1) parsing. Section 3.4.1 introduces LR(1) parsers and their operation. Section 3.4.2 presents an algorithm to build the tables that encode an LR(1) parser. Finally, Section 3.4.3 shows what happens when the construction is applied to a non-LR(1) grammar.

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*List*)
5		|	( )

(a) Parentheses Grammar

	Action Table			Goto Table	
State	eof	(	)	*List*	*Pair*
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 7	s 8	5	6
4	r 2	r 2			
5		s 7	s 10		9
6		r 3	r 3		
7		s 7	s 12	11	6
8	r 5	r 5			
9		r 2	r 2		
10	r 4	r 4			
11		s 7	s 13		9
12		r 5	r 5		
13		r 4	r 4		

(b) *Action* and *Goto* Tables for Parentheses Grammar

■ **FIGURE 3.17** The Parentheses Grammar.

3.4.1 The LR(1) Parsing Algorithm

The critical step in a bottom-up parser, such as a table-driven LR(1) parser, is to find the next handle. Efficient handle discovery is the key to efficient bottom-up parsing. An LR(1) parser uses a handle-finding automaton, encoded into two tables, traditionally called *Action* and *Goto*. Fig. 3.16 shows a simple table-driven LR(1) parser.

The skeleton LR(1) parser interprets the *Action* and *Goto* tables to find successive handles in the reverse rightmost derivation of the input. When it finds a handle $\langle A \rightarrow \beta, k\rangle$, it reduces β at k to A in the current sentential form—the upper frontier of the partially built parse tree. Rather than build an explicit parse tree, the skeleton parser keeps a prefix of the tree's upper frontier on a stack. Each stack entry is a pair $\langle A, s\rangle$ where A is a grammar symbol and s is a parser state. The states thread together the reductions into a parse. The variable *word* holds the lookahead symbol—the first word beyond the stack's contents.

The region that follows the on-stack prefix consists entirely of terminal symbols. The first word in that region is the lookahead symbol.

To find the next handle, the LR(1) parser shifts symbols onto the stack until the automaton finds the right end of a handle at the stack top. Once it has a

Iteration	State	Word	Stack	Handle	Action
0	–	(	$ ⟨*Goal* 0⟩	– *none* –	–
1	0	(	$ ⟨*Goal* 0⟩	– *none* –	*shift 3*
2	3	)	$ ⟨*Goal* 0⟩ ⟨(3⟩	– *none* –	*shift 8*
3	8	`eof`	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨) 8⟩	( )	*reduce 5*
4	2	`eof`	$ ⟨*Goal* 0⟩ ⟨*Pair* 2⟩	*Pair*	*reduce 3*
5	1	`eof`	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩	*List*	*accept*

■ **FIGURE 3.18** States of the LR(1) Parser on ( ).

handle, $A \rightarrow \beta$, the parser reduces, replacing β with A. It pops the symbols in β and their associated states from the stack and pushes A and its new state onto the stack. The *Action* and *Goto* tables thread together actions (shifts and reduces) and states in a grammar-driven sequence to find a rightmost derivation, if one exists.

Using a stack lets the LR(1) parser make the position, k, in the handle be constant and implicit.

This simplification makes the set of handles finite and recognizable by a DFA.

To make this concrete, consider the grammar from Fig. 3.17(a), which describes the language of properly nested parentheses. Panel (b) shows the *Action* and *Goto* tables for this grammar. They encode the handle-finding automaton for the parentheses language.

Fig. 3.18 shows the sequence of actions that the parser takes on the input string "( )". Each line shows one iteration of the parser; the first line shows the parser's initial state. The final column shows the parsing action dictated by the table in that iteration.

The symbol $ on the stack represents the pair ⟨*INVALID*, *INVALID*⟩.

At the start of the first iteration, the stack lacks a handle. The *Action* table tells the parser to shift (onto the stack and move to state 3. At the start of the second iteration, the stack still lacks a handle; the *Action* table causes the parser to shift) onto the stack and move to state 8.

In the third iteration, the stack contains the handle *Pair* → ( ); its right end lies at the stack top. The *Action* table directs the parser to reduce ( ) to *Pair*. Using the state beneath *Pair* on the stack, 0, and *Pair*, the parser moves to state *Goto[0,Pair]* = *2*.

The chain of handles, (5, 3, 1), shows rightmost derivation in reverse order.

In the fourth iteration, *Pair* is atop the stack. The parser finds the handle *List* → *Pair*, reduces, and moves to state *Goto[0,List]* = *1*. Finally, in state 1, the parser finds the handle *Goal* → *List* and accepts.

The *accept* action is a special action that signals a reduction to the goal symbol with lookahead `eof`.

For the input ( ), the parser used two shifts, two reduces, and one accept. In general, LR(1) parsers take time proportional to the length of the input (one shift per word) and the length of the derivation (one reduce per derivation step). Thus, derivation length is a lower bound on parsing time.

Iteration	State	Word	Stack	Handle	Action
0	—	(	$ ⟨*Goal* 0⟩	– *none* –	—
1	0	(	$ ⟨*Goal* 0⟩	– *none* –	*shift 3*
2	3	(	$ ⟨*Goal* 0⟩ ⟨(3⟩	– *none* –	*shift 7*
3	7	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨(7⟩	– *none* –	*shift 12*
4	12	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨(7⟩ ⟨) 12⟩	( )	*reduce 5*
5	6	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨*Pair* 6⟩	*Pair*	*reduce 3*
6	5	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨*List* 5⟩	– *none* –	*shift 10*
7	10	(	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨*List* 5⟩ ⟨) 10⟩	(*List*)	*reduce 4*
8	2	(	$ ⟨*Goal* 0⟩ ⟨*Pair* 2⟩	*Pair*	*reduce 3*
9	1	(	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩	– *none* –	*shift 3*
10	3	)	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩ ⟨(3⟩	– *none* –	*shift 8*
11	8	`eof`	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩ ⟨(3⟩ ⟨) 8⟩	( )	*reduce 5*
12	4	`eof`	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩ ⟨*Pair* 4⟩	*List Pair*	*reduce 2*
13	1	`eof`	$ ⟨*Goal* 0⟩ ⟨*List* 1⟩	*List*	*accept*

■ **FIGURE 3.19** States of the LR(1) Parser on (( )) ( ).

Fig. 3.19 shows the parser's behavior for the input "(( )) ( )." The parser performs six shifts, six reduces, and one accept. Fig. 3.20 shows the state of the partially built parse tree at the end of each iteration, from 1 to 12. The accept action could add a node for the *Goal* symbol if needed. The top of each drawing shows an iteration number and a gray bar that contains the partial parse tree's upper frontier—the symbols on the stack in an LR(1) parser.

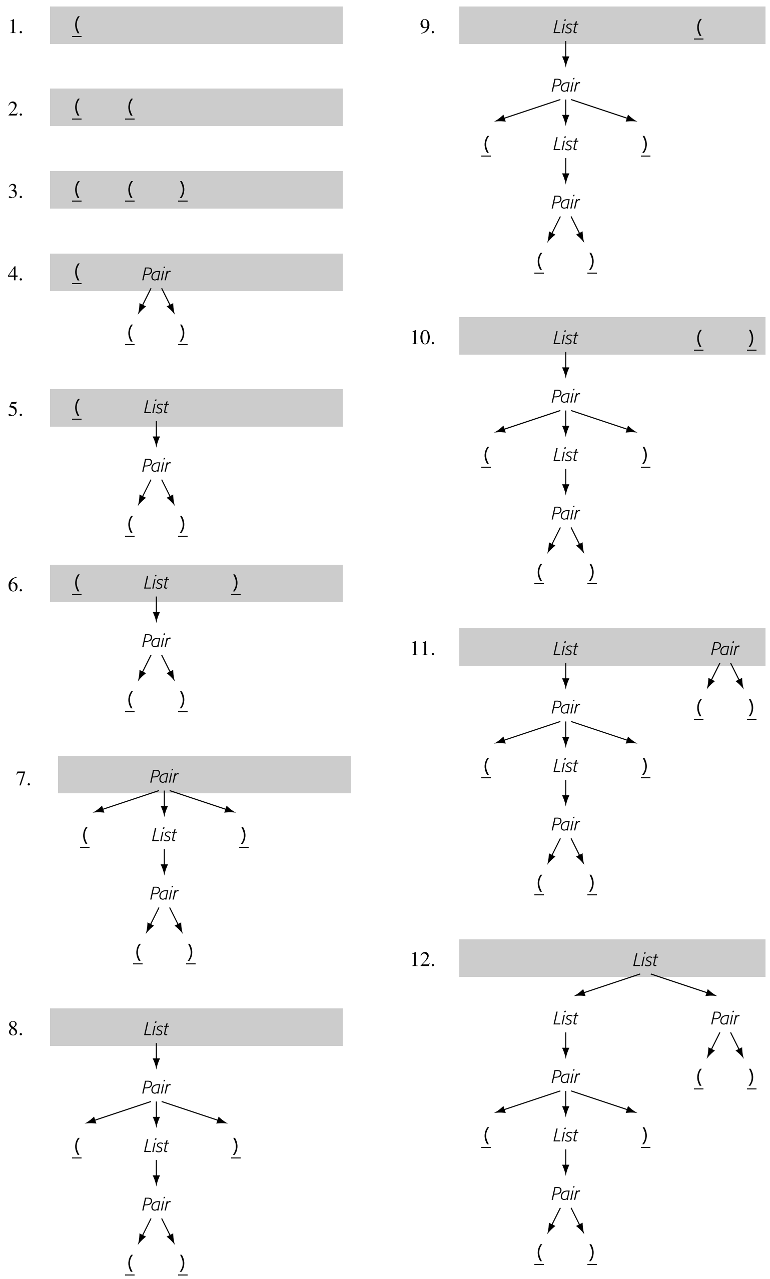
Handle Finding

The LR(1) parser's ability to find handles on the stack is the key to its operation. Consider the parser's actions on the string "( )", shown in Fig. 3.18. Iterations 1 and 2 have no handle on top of the stack, so it shifts. In each of iterations 3, 4, and 5, it finds a handle atop the stack.

Iteration 3 (state 8): The stack top ( ) forms the handle for rule 5. *Action*[8,`eof`] = *Action*[8,(] = *reduce 5*.

Iteration 4 (state 2): The stack top *Pair* forms the handle for rule 3. *Action*[2,`eof`] = *Action*[2,(] = *reduce 3*.

Iteration 5 (state 1): The stack top *List* forms the handle for rule 1 if the lookahead is `eof`. *Action*[1,`eof`] = *accept*. With other lookaheads, *List* is not a handle.



■ **FIGURE 3.20** The Sequence of Partial Parse Trees Built for (( )) ( ).

Between these two examples, the parser recognized ( ) as a handle three times. Each case behaved differently, based on the prior left context encoded in the state under ( ) on the stack. Comparing these three situations exposes how the stacked states encode left context and control the future direction of the parse.

With the first example, in Fig. 3.18, the parser was in state 8 when it found the handle ( ), with a lookahead of `eof`. The state underneath the handle was 0, and *Goto*[0,*Pair*] = 2. In state 2, a lookahead of `eof` led to a reduction by rule 3 followed by an accept.

If, instead of `eof`, the lookahead in state 2 had been), the parser would have thrown an error. A lookahead of (would have led to a reduction by rule 3—setting up to recognize more nested parentheses.

The second example, in Fig. 3.19, finds a handle for ( ) twice. The first handle occurs in iteration 4. The parser is in state 12 with a lookahead of). It has previously shifted (, (, and) onto the stack. *Action*[12,)] = *reduce 5*. The state below ( ) is 3, and *Goto*[3,*Pair*] = 6, a state in which further)'s are legal. The handle ( ) also occurs in iteration 11. Here, state 1 is beneath ( ) on the stack and *Goto*[1,*Pair*] = 4. In state 4, a lookahead of either `eof` or (triggers a reduction of *List Pair* to *List*, while a lookahead of) is an error.

The *Action* and *Goto* tables, along with the stack, cause the parser to track prior left context and let it take different actions based on that context. This context, in turn, lets the parser handle correctly each of the three instances in which it found a handle for ( ). We will revisit this issue when we examine the construction of *Action* and *Goto*.

Parsing an Erroneous Input String

To see how an LR(1) parser discovers a syntax error, consider the sequence of actions that it takes on the string "( ))".

Iteration	State	Word	Stack	Handle	Action
0	—	(	$ ⟨*Goal* 0⟩	*– none –*	—
1	0	(	$ ⟨*Goal* 0⟩	*– none –*	*shift 3*
2	3	)	$ ⟨*Goal* 0⟩ ⟨(3⟩	*– none –*	*shift 8*
3	8	)	$ ⟨*Goal* 0⟩ ⟨(3⟩ ⟨) 8⟩	*– none –*	*error*

The initial state and the first two iterations of the parse match the first example, "( )". The parser shifts (and). In the third iteration of the while loop, the parser reads *Action*[8,)] and finds an invalid entry. The invalid entry causes the parser to throw an error.

The LR(1) parser detects syntax errors with a simple mechanism: it finds an invalid table entry. It detects the error as soon as possible, before it reads a word beyond those needed to prove the input erroneous. This property localizes the error to a specific point in the input. Using the available context and knowledge of the grammar, an LR(1) parser can provide good diagnostic error messages.

Error localization is a particular strength of LR(1) parsers.

Using LR Parsers

The key to LR parsing lies in the construction of the *Action* and *Goto* tables. The tables encode all of the legal reduction sequences that can arise in a reverse rightmost derivation for the given grammar. While the number of such sequences is huge, the grammar itself constrains the order in which reductions can occur.

The compiler writer can build *Action* and *Goto* tables by hand. However, the table-construction algorithm requires scrupulous bookkeeping; it is a prime example of the kind of task that should be automated and relegated to a computer. Programs that automate this construction are widely available. The next section presents one algorithm that can be used to construct LR(1) parse tables.

With an LR(1) parser generator, the compiler writer's role is to define the grammar and to ensure that the grammar has the LR(1) property. In practice, the parser generator identifies the productions that introduce ambiguity or that need more than a one word lookahead to distinguish a shift action from a reduce action. As we study the table-construction algorithm, we will see how those problems arise, how to cure them, and how to understand the kinds of diagnostic information that LR(1) parser generators produce.

Using More Lookahead

The ideas that underlie LR(1) parsers actually define a family of parsers that vary in the amount of lookahead that they use. An LR(k) parser uses, at most, k lookahead symbols. Additional lookahead allows an LR(2) parser to recognize a larger set of grammars than an LR(1) parser. Almost paradoxically, the added lookahead does not increase the set of languages that these parsers can recognize. LR(1) parsers accept the same set of languages as LR(k) parsers for $k > 1$. The LR(1) grammar for a language may be more complex than an LR(2) or LR(3) grammar.

3.4.2 Building LR(1) Tables

To construct the *Action* and *Goto* tables for a grammar, an LR(1) parser generator builds a model of the handle-recognizing automaton and uses that model to fill in the tables. The model, called the *canonical collection of sets of LR(1) items*, represents all of the possible states of the parser, as well as the transitions between those states. The process is reminiscent of the subset construction from Section 2.4.3.

1	*Goal*	→	*List*
2	*List*	→	*List Pair*
3		|	*Pair*
4	*Pair*	→	(*List*)
5		|	( )

The Parentheses Grammar

To illustrate the table-construction algorithm, we will use two examples. The first is the parentheses grammar from Fig. 3.17(a), repeated in the margin. It is small enough to use as a running example, but large enough to exhibit some of the complexities of the process. The second example, in Section 3.4.3, is an abstracted version of the classic `if-then-else` ambiguity. The construction fails on this grammar because the grammar is ambiguous. The example highlights the situations that lead to failures in the table-construction process.

LR(1) Items

LR(1) item
$[A \rightarrow \beta \bullet \gamma, \mathtt{a}]$ where $A \rightarrow \beta\gamma$ is a grammar production, $\bullet$ represents the position of the parser's stacktop, and $\mathtt{a}$ is a terminal symbol in the grammar

In an LR(1) parser, the *Action* and *Goto* tables encode information about handles and potential handles at each step in the parse. The table-construction algorithm uses LR(1) items to represent handles, potential handles, and the associated lookahead symbols. An LR(1) item $[A \rightarrow \beta \bullet \gamma, \mathtt{a}]$ consists of a production $A \rightarrow \beta\gamma$; a placeholder, $\bullet$, that indicates the position of the stacktop in the production's right-hand side; and a specific terminal symbol, $\mathtt{a}$, as a lookahead symbol.

The table-construction algorithm builds the canonical collection of sets of LR(1) items, $\mathcal{CC} = \{ \text{CC}_0, \text{CC}_1, \text{CC}_2, \ldots, \text{CC}_n \}$. Each $\text{CC}_i \in \mathcal{CC}$ represents a valid configuration of the LR(1) parser, or a state of the parser. It contains one or more LR(1) items that represent handles or potential handles that correspond to the parse state. Before we delve into the table construction, more explanation of LR(1) items is needed.

For a production $A \rightarrow \beta\gamma$ and a lookahead symbol $\mathtt{a}$, the placeholder can generate three distinct items, each with its own interpretation. In each case, the presence of the item in some set CC_i in the canonical collection indicates input that the parser has seen is consistent with the occurrence of an A followed by an $\mathtt{a}$ in the grammar. The position of $\bullet$ in the item distinguishes between the three cases.

1. $[A \rightarrow \bullet\beta\gamma, \mathtt{a}]$ indicates that an A would be valid and that recognizing a β next would be one step toward discovering an A. We call such an item a *possibility*; it represents a possible completion for the input already seen.

$[\mathit{Goal} \rightarrow \bullet\, \mathit{List}, \texttt{eof}]$

$[\mathit{Goal} \rightarrow \mathit{List}\, \bullet, \texttt{eof}]$

$[\mathit{List} \rightarrow \bullet\, \mathit{Pair}, \texttt{eof}]$

$[\mathit{List} \rightarrow \mathit{Pair}\, \bullet, \texttt{eof}]$

$[\mathit{List} \rightarrow \bullet\, \mathit{Pair}, \underline{(}]$

$[\mathit{List} \rightarrow \mathit{Pair}\, \bullet, \underline{(}]$

$[\mathit{List} \rightarrow \bullet\, \mathit{Pair}, \underline{)}]$

$[\mathit{List} \rightarrow \mathit{Pair}\, \bullet, \underline{)}]$

$[\mathit{List} \rightarrow \bullet\, \mathit{List}\ \mathit{Pair}, \texttt{eof}]$

$[\mathit{List} \rightarrow \mathit{List}\, \bullet\, \mathit{Pair}, \texttt{eof}]$

$[\mathit{List} \rightarrow \mathit{List}\ \mathit{Pair}\, \bullet, \texttt{eof}]$

$[\mathit{List} \rightarrow \bullet\, \mathit{List}\ \mathit{Pair}, \underline{(}]$

$[\mathit{List} \rightarrow \mathit{List}\, \bullet\, \mathit{Pair}, \underline{(}]$

$[\mathit{List} \rightarrow \mathit{List}\ \mathit{Pair}\, \bullet, \underline{(}]$

$[\mathit{List} \rightarrow \bullet\, \mathit{List}\ \mathit{Pair}, \underline{)}]$

$[\mathit{List} \rightarrow \mathit{List}\, \bullet\, \mathit{Pair}, \underline{)}]$

$[\mathit{List} \rightarrow \mathit{List}\ \mathit{Pair}\, \bullet, \underline{)}]$

$[\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \mathit{List}\ \underline{)}, \texttt{eof}]$

$[\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \mathit{List}\ \underline{)}, \texttt{eof}]$

$[\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\, \bullet\, \underline{)}, \texttt{eof}]$

$[\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \underline{)}\, \bullet, \texttt{eof}]$

$[\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \mathit{List}\ \underline{)}, \underline{(}]$

$[\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \mathit{List}\ \underline{)}, \underline{(}]$

$[\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\, \bullet\, \underline{)}, \underline{(}]$

$[\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \underline{)}\, \bullet, \underline{(}]$

$[\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \mathit{List}\ \underline{)}, \underline{)}]$

$[\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \mathit{List}\ \underline{)}, \underline{)}]$

$[\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\, \bullet\, \underline{)}, \underline{)}]$

$[\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \underline{)}\, \bullet, \underline{)}]$

$[\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}]$

$[\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \underline{)}, \texttt{eof}]$

$[\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\, \bullet, \texttt{eof}]$

$[\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}]$

$[\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \underline{)}, \underline{(}]$

$[\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\, \bullet, \underline{(}]$

$[\mathit{Pair} \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{)}]$

$[\mathit{Pair} \rightarrow \underline{(}\, \bullet\, \underline{)}, \underline{)}]$

$[\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\, \bullet, \underline{)}]$

■ **FIGURE 3.21** LR(1) Items for the Parentheses Grammar.

2. $[A \rightarrow \beta \bullet \gamma, \texttt{a}]$ indicates that the parser has progressed from the state $[A \rightarrow \bullet \beta\gamma, \texttt{a}]$ by recognizing β. One valid next step would be to recognize a γ. We call such an item *partially complete.*
3. $[A \rightarrow \beta\gamma \bullet, \texttt{a}]$ indicates that the parser has found $\beta\gamma$ in a context where an A followed by an $\texttt{a}$ would be valid. If the lookahead symbol is $\texttt{a}$, then the item is a handle and the parser can reduce $\beta\gamma$ to A. We call such an item *complete*.

In an LR(1) item, the symbols to the left of the placeholder • represent context from the portions of the production already recognized—left context. (The states on the stack encode a summary of that left context—in essence, the history of the parse so far.) The lookahead symbol provides right context—one symbol of right context. When the parser finds itself in a state that includes $[A \rightarrow \beta\gamma \bullet, \texttt{a}]$ and the lookahead is $\texttt{a}$, it has a handle and should reduce $\beta\gamma$ to A.

Fig. 3.21 shows the complete set of LR(1) items generated by the parentheses grammar. The first two items, in the upper left corner, deserve particular notice. The first, $[\mathit{Goal} \rightarrow \bullet\, \mathit{List}, \texttt{eof}]$, represents the parser's initial state—looking for a string that reduces to *Goal*, followed by $\texttt{eof}$. Every parse begins in this state. The second, $[\mathit{Goal} \rightarrow \mathit{List}\, \bullet, \texttt{eof}]$, represents the parser's desired final state—finding a string that reduces to *Goal*, followed by $\texttt{eof}$. This item represents every successful parse. All of the possible parses result from stringing together parser states in a grammar-directed way, beginning with a state that contains $[\mathit{Goal} \rightarrow \bullet\, \mathit{List}, \texttt{eof}]$ and ending with a state that contains $[\mathit{Goal} \rightarrow \mathit{List}\, \bullet, \texttt{eof}]$.

Constructing the Canonical Collection

To build the canonical collection of sets of LR(1) items, $\mathcal{CC}$, a parser generator starts from the parser's initial state, [*Goal* → • *List*, `eof`], and constructs a model of all the potential transitions that can occur. The algorithm represents each possible configuration, or state, of the parser as a set of LR(1) items. The algorithm applies two operations to these sets of LR(1) items: taking a closure and computing a transition.

- The *closure* function completes a state; given some core set of items, it adds to that set any related LR(1) items that they imply. For example, anywhere that *Goal* → *List* is legal, the productions that derive a *List* are legal, too. Thus, the item [*Goal* → • *List*, `eof`] implies both [*List* → • *List Pair*, `eof`] and [*List* → • *Pair*, `eof`]. *Closure* finds all such items and adds them to the state.
- The *goto* function models the effect of a transition from some state s on a grammar symbol x. To do so, the algorithm examines the set of LR(1) items in s. It finds each item where • precedes x, moves the • past the x, and places this new item into the set that *goto* returns as its result. Finally, it uses *closure* to complete the new state.

To make the start symbol easy to find, we require that the grammar have a unique start symbol that does not appear on the right-hand side of any production. In the parentheses grammar, that symbol is *Goal*.

If a grammar has multiple productions for the start symbol, each of them adds an item to the core of CC_0.

The item [*Goal* → • *List*, `eof`] represents the parser's initial state for the parentheses grammar; every valid parse recognizes *Goal* followed by `eof`. This item forms the core of CC_0, the first state in $\mathcal{CC}$.

Computing Closure

To compute the complete initial state of the parser, CC_0, from its core, the algorithm must add to the core all of the items implied by the items in the core. Fig. 3.22(a) shows the algorithm, *closure*, for this computation. It iterates over all the items in set s. If the placeholder • in an item immediately precedes some nonterminal C, then *closure* must add one or more items for each production that can derive C. Closure places the • at the initial position of each item that it builds this way.

Here, β and δ represent possible symbols before or after follow *C*. Either β or δ may be empty.

The rationale for *closure* is clear. If $[A \rightarrow \beta \bullet C \delta, \mathtt{a}] \in s$ and the parser reccognizes a C followed by $\delta\mathtt{a}$, it should reduce $\beta\, C\, \delta$ to A. Closure adds an item to s for each such possibility.

In our experience, this use of FIRST(δa) is the point in the LR(1) table construction where a human is most to likely make a mistake.

For some production $C \rightarrow \gamma$, *closure* builds an item for each each terminal symbol in FIRST(δa). It inserts the placeholder symbol • before γ and adds the appropriate lookahead symbol.

```
closure(s)
    while (s is still changing) do
        for each item $[A \rightarrow \beta \bullet C\,\delta, \texttt{a}] \in s$ do
            $lookahead \leftarrow \delta\texttt{a}$
            for each production $C \rightarrow \gamma \in P$ do
                for each $\texttt{b} \in \text{FIRST}(lookahead)$ do
                    $s \leftarrow s \cup \{[C \rightarrow \bullet\,\gamma, \texttt{b}]\}$
    return $s$
```

(a) The *Closure* Function

```
goto(s, x)
    $t \leftarrow \emptyset$
    for each item $i \in s$ do
        if $i$ is $[\alpha \rightarrow \beta \bullet x\,\delta, \texttt{a}]$ then
            $t \leftarrow t \cup \{[\alpha \rightarrow \beta\,x \bullet \delta, \texttt{a}]\}$
    return closure(t)
```

(b) The *Goto* Function

■ **FIGURE 3.22** Support Functions for the LR(1) Table Construction.

For the parentheses grammar, the construction creates CC_0 by building a set with the initial item, { [*Goal* → • *List*, eof] }, and computing its *closure*:

$$CC_0 = \left\{ \begin{array}{l} [Goal \rightarrow \bullet\ List, \texttt{eof}],\ [List \rightarrow \bullet\ List\ Pair, \texttt{eof}], \\ [List \rightarrow \bullet\ List\ Pair, \underline{(}],\ [List \rightarrow \bullet\ Pair, \texttt{eof}], \\ [List \rightarrow \bullet\ Pair, \underline{(}],\ [Pair \rightarrow \bullet\ \underline{(}\ List\ \underline{)}, \texttt{eof}], \\ [Pair \rightarrow \bullet\ \underline{(}\ List\ \underline{)}, \underline{(}],\ [Pair \rightarrow \bullet\ \underline{(}\ \underline{)}, \texttt{eof}], \\ [Pair \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

Closure is another example of a fixed-point computation. The triply-nested loop either adds items to s or leaves s intact. It never removes an item from s. Since the set of LR(1) items is finite, this loop must halt. The triply nested loop looks expensive. However, close examination reveals that each item in s needs to be processed only once. A worklist version of the algorithm can capitalize on that fact.

Computing Goto

The second key operation in the construction is the function *goto*. *Goto* takes as input a set that models a parser state, one of the $CC_i \in \mathcal{CC}$, and a grammar symbol x. From CC_i and x, it computes a model of the parser state that would result from recognizing an x in the state CC_i.

The *goto* function, shown in Fig. 3.22(b), takes a set of LR(1) items s and a grammar symbol x and returns a new set of LR(1) items. It iterates over the items in s. When it finds an item in which the • immediately precedes x, it creates a new item by moving the • rightward past x. This new item represents the parser's configuration after recognizing x. *Goto* places these new items in a new set, takes its *closure* to complete the parser state, and returns that new state.

Given CC_0, shown earlier, the construction can derive the state that the parser will reach after an initial $\underline{(}$ by computing *goto*(CC_0,$\underline{(}$). The inner loop in *goto*

```
$\text{CC}_0 \leftarrow \emptyset$
for each production of the form $Goal \rightarrow \alpha$ do
    $\text{CC}_0 \leftarrow \text{CC}_0 \cup \{\, [\, Goal \rightarrow \bullet\, \alpha,\ \texttt{eof}\,]\, \}$
$\text{CC}_0 \leftarrow closure(\,\text{CC}_0)$
$\mathcal{CC} \leftarrow \{\, \text{CC}_0 \,\}$

while (new sets are still being added to $\mathcal{CC}$) do
    for each unmarked set $\text{CC}_i \in \mathcal{CC}$ do
        mark $\text{CC}_i$ as processed
        for each $x$ following a $\bullet$ in an item in $\text{CC}_i$ do
            $temp \leftarrow goto(\text{CC}_i, x)$
            if $temp \notin \mathcal{CC}$ then
                $\mathcal{CC} \leftarrow \mathcal{CC} \cup \{temp\}$
            record transition from $\text{CC}_i$ to $temp$ on $x$
```

■ **FIGURE 3.23** The Algorithm to Build the Canonical Collection of Sets of LR(1) Items.

finds four items that contain "$\bullet\ \underline{(}$". From each of those items, *goto* creates a new item with the $\bullet$ moved past the $\underline{(}$. These items form the *core* of the new set—the first four items. *Goto* then invokes *closure* to complete the set. In the construction, this set becomes CC_3.

$$\text{CC}_3 = \left\{ \begin{array}{l} [Pair \rightarrow \underline{(}\ \bullet\ List\ \underline{)},\ \texttt{eof}]\ [Pair \rightarrow \underline{(}\ \bullet\ List\ \underline{)},\ \underline{(}] \\ [Pair \rightarrow \underline{(}\ \bullet\ \underline{)},\ \texttt{eof}]\ [Pair \rightarrow \underline{(}\ \bullet\ \underline{)},\ \underline{(}] \\ [List \rightarrow \bullet\ List\ Pair,\ \underline{(}]\ [List \rightarrow \bullet\ List\ Pair,\ \underline{)}] \\ [List \rightarrow \bullet\ Pair,\ \underline{(}]\ [List \rightarrow \bullet\ Pair,\ \underline{)}]\ [Pair \rightarrow \bullet\ \underline{(}\ List\ \underline{)},\ \underline{(}] \\ [Pair \rightarrow \bullet\ \underline{(}\ List\ \underline{)},\ \underline{)}]\ [Pair \rightarrow \bullet\ \underline{(}\ \underline{)},\ \underline{(}]\ [Pair \rightarrow \bullet\ \underline{(}\ \underline{)},\ \underline{)}] \end{array} \right\}$$

Goto created the first four items directly. *Closure* added the rest.

The Algorithm

To construct the canonical collection of sets of LR(1) items, $\mathcal{CC}$, the algorithm computes the initial set, CC_0, and then systematically finds sets that represent each possible parser state reachable from CC_0. It repeats this process with each new set, until it has built the set of all reachable parser states. Fig. 3.23 shows the algorithm.

To compute the initial state, the algorithm constructs an LR(1) item that represents each production that has the goal symbol on its left-hand side. It computes CC_0 as the *closure* of the union of these items.

To find all the states reachable from CC_0, the algorithm finds each symbol x that follows the placeholder in an item in CC_0. For each such x, it computes $goto(\text{CC}_0, x)$. If that set is not already in $\mathcal{CC}$, it adds the set to $\mathcal{CC}$. It also records the transition on x from CC_0 to $goto(\text{CC}_0, x)$.

Iteration	Set	*Goal*	*List*	*Pair*	(	)
1	CC_0	∅	CC_1	CC_2	CC_3	∅
2	CC_1	∅	∅	CC_4	CC_3	∅
	CC_2	∅	∅	∅	∅	∅
	CC_3	∅	CC_5	CC_6	CC_7	CC_8
3	CC_4	∅	∅	∅	∅	∅
	CC_5	∅	∅	CC_9	CC_7	CC_{10}
	CC_6	∅	∅	∅	∅	∅
	CC_7	∅	CC_{11}	CC_6	CC_7	CC_{12}
	CC_8	∅	∅	∅	∅	∅
4	CC_9	∅	∅	∅	∅	∅
	CC_{10}	∅	∅	∅	∅	∅
	CC_{11}	∅	∅	CC_9	CC_7	CC_{13}
	CC_{12}	∅	∅	∅	∅	∅
5	CC_{13}	∅	∅	∅	∅	∅

■ **FIGURE 3.24** Trace of the LR(1) Construction on the Parentheses Grammar.

To build the complete canonical collection, the algorithm repeats this process with each successive set in the collection, CC_i. The process is reminiscent of the subset construction (see Section 2.6). To ensure that the algorithm processes each set CC_i exactly once, it uses a simple marking scheme. It creates each set in an unmarked condition and marks the set as it is processed, which drastically reduces the number of times that it invokes *goto* and *closure*.

A worklist formulation would achieve the same effect.

This construction is a fixed-point computation. The canonical collection $\mathcal{CC}$ is a subset of the powerset of the LR(1) items. The while loop is monotonic in the size of $\mathcal{CC}$; it adds new sets to $\mathcal{CC}$ but never removes one. If the set of LR(1) items has n elements, then $\mathcal{CC}$ can grow no larger than 2^n items. Since 2^n is finite, the computation must halt.

This upper bound on the size of the canonical collection is quite loose. For example, the parentheses grammar has 38 LR(1) items, but it produces a $\mathcal{CC}$ that has only 14 sets. The upper bound would be 2^{38}, a much larger number. For more complex grammars, the size of $\mathcal{CC}$ is a concern, primarily because the *Action* and *Goto* tables grow with $|\mathcal{CC}|$. As described in Section 3.6, both the compiler writer and the parser-generator writer can take steps to reduce the size of those tables.

$$\mathrm{CC}_0 = \left\{ \begin{array}{l} [\mathit{Goal} \rightarrow \bullet\ \mathit{List}, \texttt{eof}] \\ [\mathit{List} \rightarrow \bullet\ \mathit{List}\ \mathit{Pair}, \texttt{eof}]\ [\mathit{List} \rightarrow \bullet\ \mathit{List}\ \mathit{Pair}, \underline{(}]\ [\mathit{List} \rightarrow \bullet\ \mathit{Pair}, \texttt{eof}]\ [\mathit{List} \rightarrow \bullet\ \mathit{Pair}, \underline{(}] \\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \texttt{eof}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \texttt{eof}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

$$\mathrm{CC}_1 = \left\{ \begin{array}{l} [\mathit{Goal} \rightarrow \mathit{List}\ \bullet, \texttt{eof}]\ [\mathit{List} \rightarrow \mathit{List}\ \bullet\ \mathit{Pair}, \texttt{eof}]\ [\mathit{List} \rightarrow \mathit{List}\ \bullet\ \mathit{Pair}, \underline{(}] \\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \texttt{eof}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \texttt{eof}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

$$\mathrm{CC}_2 = \left\{ [\mathit{List} \rightarrow \mathit{Pair}\ \bullet, \texttt{eof}]\ [\mathit{List} \rightarrow \mathit{Pair}\ \bullet, \underline{(}] \right\}$$

$$\mathrm{CC}_3 = \left\{ \begin{array}{l} [\mathit{Pair} \rightarrow \underline{(}\ \bullet\ \mathit{List}\ \underline{)}, \texttt{eof}]\ [\mathit{Pair} \rightarrow \underline{(}\ \bullet\ \mathit{List}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \underline{(}\ \bullet\ \underline{)}, \texttt{eof}]\ [\mathit{Pair} \rightarrow \underline{(}\ \bullet\ \underline{)}, \underline{(}] \\ [\mathit{List} \rightarrow \bullet\ \mathit{List}\ \mathit{Pair}, \underline{(}]\ [\mathit{List} \rightarrow \bullet\ \mathit{List}\ \mathit{Pair}, \underline{)}]\ [\mathit{List} \rightarrow \bullet\ \mathit{Pair}, \underline{(}]\ [\mathit{List} \rightarrow \bullet\ \mathit{Pair}, \underline{)}] \\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{)}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$\mathrm{CC}_4 = \left\{ [\mathit{List} \rightarrow \mathit{List}\ \mathit{Pair}\ \bullet, \texttt{eof}]\ [\mathit{List} \rightarrow \mathit{List}\ \mathit{Pair}\ \bullet, \underline{(}] \right\}$$

$$\mathrm{CC}_5 = \left\{ \begin{array}{l} [\mathit{List} \rightarrow \mathit{List}\ \bullet\ \mathit{Pair}, \underline{(}]\ [\mathit{List} \rightarrow \mathit{List}\ \bullet\ \mathit{Pair}, \underline{)}]\ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \bullet\ \underline{)}, \texttt{eof}]\ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \bullet\ \underline{)}, \underline{(}] \\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{)}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$\mathrm{CC}_6 = \left\{ [\mathit{List} \rightarrow \mathit{Pair}\ \bullet, \underline{(}]\ [\mathit{List} \rightarrow \mathit{Pair}\ \bullet, \underline{)}] \right\}$$

$$\mathrm{CC}_7 = \left\{ \begin{array}{l} [\mathit{Pair} \rightarrow \underline{(}\ \bullet\ \mathit{List}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \underline{(}\ \bullet\ \mathit{List}\ \underline{)}, \underline{)}]\ [\mathit{Pair} \rightarrow \underline{(}\ \bullet\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \underline{(}\ \bullet\ \underline{)}, \underline{)}] \\ [\mathit{List} \rightarrow \bullet\ \mathit{List}\ \mathit{Pair}, \underline{(}]\ [\mathit{List} \rightarrow \bullet\ \mathit{List}\ \mathit{Pair}, \underline{)}]\ [\mathit{List} \rightarrow \bullet\ \mathit{Pair}, \underline{(}]\ [\mathit{List} \rightarrow \bullet\ \mathit{Pair}, \underline{)}] \\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{)}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$\mathrm{CC}_8 = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\ \bullet, \texttt{eof}]\ [\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\ \bullet, \underline{(}] \right\}$$

$$\mathrm{CC}_9 = \left\{ [\mathit{List} \rightarrow \mathit{List}\ \mathit{Pair}\ \bullet, \underline{(}]\ [\mathit{List} \rightarrow \mathit{List}\ \mathit{Pair}\ \bullet, \underline{)}] \right\}$$

$$\mathrm{CC}_{10} = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \underline{)}\ \bullet, \texttt{eof}]\ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \underline{)}\ \bullet, \underline{(}] \right\}$$

$$\mathrm{CC}_{11} = \left\{ \begin{array}{l} [\mathit{List} \rightarrow \mathit{List}\ \bullet\ \mathit{Pair}, \underline{(}]\ [\mathit{List} \rightarrow \mathit{List}\ \bullet\ \mathit{Pair}, \underline{)}]\ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \bullet\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \bullet\ \underline{)}, \underline{)}] \\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \mathit{List}\ \underline{)}, \underline{)}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{(}]\ [\mathit{Pair} \rightarrow \bullet\ \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}$$

$$\mathrm{CC}_{12} = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\ \bullet, \underline{(}]\ [\mathit{Pair} \rightarrow \underline{(}\ \underline{)}\ \bullet, \underline{)}] \right\}$$

$$\mathrm{CC}_{13} = \left\{ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \underline{)}\ \bullet, \underline{(}]\ [\mathit{Pair} \rightarrow \underline{(}\ \mathit{List}\ \underline{)}\ \bullet, \underline{)}] \right\}$$

■ **FIGURE 3.25** The Canonical Collection of Sets of LR(1) Items for the Parentheses Grammar.

Building the Canonical Collection

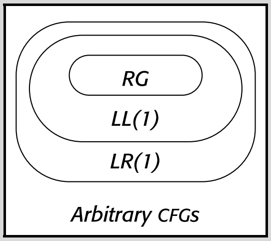
As a first complete example, consider the problem of building $\mathcal{CC}$ for the parentheses grammar. Fig. 3.24 summarizes the steps the algorithm takes. Fig. 3.25 shows the individual sets that it creates.

The algorithm first computes *closure*([*Goal* → • *List*, eof]), which becomes CC_0. It then begins to build possible parser configurations. From iteration 1 in Fig. 3.24, we see that *goto*(CC_0, *List*) creates CC_1, *goto*(CC_0, *Pair*) creates

CLASSES OF CONTEXT-FREE GRAMMARS

We can partition the universe of context-free grammars into a hierarchy based on the difficulty of parsing the grammars. This chapter distinguishes between four kinds of grammars: arbitrary CFGs, LR(1) grammars, LL(1) grammars, and regular grammars (RGs). These sets nest as shown below.

Arbitrary CFGs require more time to parse than do the more restricted LR(1) or LL(1) grammars. Earley's algorithm, for example, parses arbitrary CFGs, but it has a worst case time bound of $O(n^3)$, where *n* is the number of words in the input. Historically, compiler writers have considered Earley's algorithm too expensive for use in practical applications.



The set of LR(1) grammars includes a large subset of the unambiguous CFGs. LR(1) grammars can be parsed, bottom-up, in a linear, left-to-right scan, with a one-word lookahead. Tools that build LR(1) parsers are widely available.

The set of LL(1) grammars is an important subset of the LR(1) grammars. LL(1) grammars can be parsed, top-down, in a linear, left-to-right scan, with a one-word lookahead. Many tools exist to build LL(1) parsers; the grammars are also suitable for hand-coded recursive-descent parsers. Many programming languages can be specified with an LL(1) grammar.

Regular grammars (RGs) are a subset of CFGs where the form of productions is restricted to either $A \rightarrow \texttt{a}$ or $A \rightarrow \texttt{a}\,B$, with $A, B \in NT$ and $\texttt{a} \in T$. Regular grammars encode precisely the same languages as regular expressions.

Almost all programming-language constructs can be written in LR(1) or LL(1) form. Thus, most compilers use a parser based on one of these two classes of CFGs. Some constructs fall in the gap between LR(1) and LL(1); they do not appear to be particularly useful. (See Waite's grammar in Exercise 3.10.)

cc_2, and *goto*(cc_0, $\underline{(}$) creates cc_3. The other grammar symbols, *Goal* and $\underline{)}$, produce an empty set. In each set cc_i shown in Fig. 3.25, the first line contains the core items; subsequent lines contain the items added by *closure*.

Later iterations fill in the rest of $\mathcal{CC}$, as shown in Figs. 3.24 and 3.25. The construction halts when it fails to add new sets to $\mathcal{CC}$.

Filling in the Tables

Given the canonical collection, $\mathcal{CC}$, for a grammar, the parser generator can fill in the *Action* and *Goto* tables by iterating over the $cc_i \in \mathcal{CC}$. Each cc_i becomes a parser state. Its items dictate where shifts and reduces appear in

```
for each $cc_i \in \mathcal{CC}$ do
    for each item $I \in cc_i$ do
        if $I$ is $[A \rightarrow \beta \bullet c\,\gamma, a]$ and goto($cc_i$,c) = $cc_j$ then
            Action[i,c] ← "shift j"
        else if $I$ is $[A \rightarrow \beta \bullet, a]$ then
            Action[i,a] ← "reduce $A \rightarrow \beta$"
        else if $I$ is $[Goal \rightarrow \beta \bullet, eof]$ then
            Action[i, eof] ← "accept"
    for each $n \in NT$ do
        if goto($cc_i$,n) = $cc_j$ then
            Goto[i,n] ← j
```

■ **FIGURE 3.26** LR(1) Table-Filling Algorithm.

the corresponding *Action* table row. The state transitions in *Action* and *Goto* come from the transitions recorded during construction of the canonical collection, as shown in Fig. 3.24.

Three kinds of items generate entries in the *Action* table:

As before, either β or δ, or both, can be empty.

1. An item of the form $[A \rightarrow \beta \bullet c\,\delta, a]$ indicates that finding the terminal symbol c would be a valid next step in finding the nonterminal *A*. It generates a *shift* item in the column for c in the current state. It finds the recognizer's next state in the trace under cc_i and c.
2. An item of the form $[A \rightarrow \beta \bullet, a]$ indicates that the parser has found a β. If the lookahead is a, then the item is a handle. Thus, it generates a *reduce* item for the production $A \rightarrow \beta$ in the column for a in the current state.
3. An item of the form $[Goal \rightarrow \beta \bullet, eof]$, where *Goal* is the start symbol, indicates an accepting state. The parser has reduced the input stream to *Goal* and the lookahead symbol is eof. This item generates an *accept* action in eof's column in the current state.

The table-filling actions can be integrated into the construction of $\mathcal{CC}$.

The algorithm in Fig. 3.26 makes this concrete. For an LR(1) grammar, it should uniquely define the entries in the *Action* and *Goto* tables.

To fill in the *Goto* table, the algorithm does not need to recompute *goto*(cc_i,n). Instead, it can consult the transitions that it recorded during the construction of $\mathcal{CC}$. For the example, that information is recorded in the trace shown in Fig. 3.24.

The table-filling algorithm ignores items where the • precedes a nonterminal symbol. Shift actions are generated when • precedes a terminal. Reduce and accept actions are generated when • is at the right end of the production. What if cc_i contains an item $[A \rightarrow \beta \bullet \gamma \delta, a]$, where $\gamma \in NT$? This item does not generate any table entries, but its presence in the set forces *closure* to include items that generate table entries. When *closure* finds a • that immediately precedes a nonterminal symbol γ, it adds productions that have γ as their left-hand side, with a • preceding their right-hand sides. This process instantiates FIRST(γ) in CC_i. *Closure* will find each $x \in$ FIRST(γ) and add the items into CC_i to generate shift items for each such x.

The tables produced for the parentheses grammar are shown in Fig. 3.17(b) on page 122. Those tables, plus the skeleton LR(1) parser, constitute a working parser for the parentheses language.

In practice, an LR(1) parser generator must produce other tables needed by the skeleton LR(1) parser. For example, when the skeleton parser reduces by $A \rightarrow \beta$, it pops $|\beta|$ pairs from the stack and pushes A onto the stack. The table generator must produce data structures that map a production from the reduce entry in the *Action* table, say $A \rightarrow \beta$, into both $|\beta|$ and A. Other tables, such as a map from the integer representing a grammar symbol into its textual name, are needed for debugging and for diagnostic messages.

Handle Finding, Revisited

LR(1) parsers derive their efficiency from a fast handle-finding mechanism embedded in the *Action* and *Goto* tables. The canonical collection, $\mathcal{CC}$, represents a handle-finding DFA for the grammar. Fig. 3.27 shows the DFA for our example, the parentheses grammar.

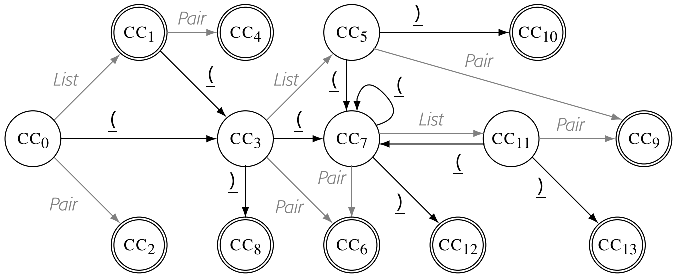
How can the LR(1) parser use a DFA to find the handles, when we know that the language of parentheses is not a regular language? The LR(1) parser relies on a simple observation: *the set of handles is finite.* The set of handles is precisely the set of complete LR(1) items—those with the placeholder • at the right end of the item's production. Any language with a finite set of sentences can be recognized by a DFA. Since the number of productions and the number of lookahead symbols are both finite, the number of complete items is finite, and the language of handles is a regular language.

The LR(1) parser makes the handle's position implicit, at stacktop. This decision greatly reduces the number of possible handles.

When the LR(1) parser executes, it interleaves two kinds of actions: shifts and reduces. The shift actions simulate steps in the handle-finding DFA. As the parser shifts each word in the input stream onto the parse stack, it also changes state in the DFA as dictated by the word's syntactic category. The reduce actions occur when the DFA reaches a final state. At that point, the skeleton parser pops the handle, and its DFA states, off the stack to reveal the state of the DFA before it began looking for the current handle. That state sits on the stack below the handle's left end.

The reduce action uses the handle's left-hand side nonterminal to take a transition in the DFA. It finds that transition in the *Goto* table, using the

Transitions made on nonterminal symbols appear in gray in Fig. 3.27.



■ **FIGURE 3.27** Handle-Finding DFA for the Parentheses Grammar.

revealed state and the nonterminal; pushes that new state on the stack; and then pushes the nonterminal. In effect, the LR(1) skeleton parser uses the reduce state actions to simulate recursive invocations of the handle-finding DFA for each new subgoal in the parse.

The LR(1) parser alternates between handle-finding phases and reduce actions. It shifts items onto the stack until it reaches a final state in the DFA. It then reduces the handle to its left-hand side nonterminal. The reduce actions tie together successive handle-finding phases. They use the left context encoded in the revealed state to restart the handle-finding DFA in a state that reflects the just-recognized nonterminal. In the parse of (( )) ( ) shown in Fig. 3.19, the parser stacks different states for the first (than it does when a (follows another (. Those stacked states allow it to find the correct handles.

The handle-finding DFA is encoded directly in the *Action* and *Goto* tables. Each dark edge corresponds to a shift action; *shift 3* in *Action*[1, (] produces an edge from CC$_1$ to CC$_3$ labeled with (. Each gray edge corresponds to a reduce action. If i is the revealed state and $Goto[i,x] = j$, then the DFA contains a gray transition from CC$_i$ to CC$_j$ labeled with the nonterminal x. Thus, the DFA in Fig. 3.27 can be read directly from the *Action* and *Goto* tables in Fig. 3.17(b).

3.4.3 **Errors in the Table Construction**

1	*Goal*	→	*Stmt*
2	*Stmt*	→	`ifthen` *Stmt*
3		|	`ifthen` *Stmt* `else` *Stmt*
4		|	`assign`

Simplified `if-then-else` Grammar

As a second example of the LR(1) table construction, consider the ambiguous grammar for the classic `if–then–else` construct, shown in the margin. To keep the size of *CC* manageable, we have replaced the "`if` *Expr* `then`" portion of rules 2 and 3 with a single terminal symbol, `ifthen`. All of the other statements are abstracted into a single terminal symbol, `assign`. The resulting grammar retains the fundamental ambiguity of the original grammar, shown on page 93.

Iteration	Item	*Goal*	*Stmt*	`ifthen`	`else`	`assign`
1	CC_0	∅	CC_1	CC_2	∅	CC_3
2	CC_1	∅	∅	∅	∅	∅
	CC_2	∅	CC_4	CC_5	∅	CC_6
	CC_3	∅	∅	∅	∅	∅
3	CC_4	∅	∅	∅	CC_7	∅
	CC_5	∅	CC_8	CC_5	∅	CC_6
	CC_6	∅	∅	∅	∅	∅
4	CC_7	∅	CC_9	CC_2	∅	CC_3
	CC_8	∅	∅	∅	CC_{10}	∅
5	CC_9	∅	∅	∅	∅	∅
	CC_{10}	∅	CC_{11}	CC_5	∅	CC_6
6	CC_{11}	∅	∅	∅	∅	∅

■ **FIGURE 3.28** Trace of the LR(1) Construction on the `if–then–else` Grammar.

The construction of the canonical collection begins by computing CC_0 as *closure*([*Goal* → • *Stmt*, `eof`]). Fig. 3.28 shows a trace of the full construction. Fig. 3.29 shows resulting canonical collection of sets of LR(1) items. Close examination of the canonical collection for this grammar shows how the table construction algorithm manifests errors and how the table-filling algorithm discovers those errors.

The table-filling algorithm proceeds normally until it reaches CC_8:

$$CC_8 = \left\{ \begin{array}{ll} 1. & [\,\mathit{Stmt} \rightarrow \texttt{ifthen}\ \mathit{Stmt}\ \bullet,\ \texttt{eof}\,] \\ 2. & [\,\mathit{Stmt} \rightarrow \texttt{ifthen}\ \mathit{Stmt}\ \bullet,\ \texttt{else}\,] \\ 3. & [\,\mathit{Stmt} \rightarrow \texttt{ifthen}\ \mathit{Stmt}\ \bullet\ \texttt{else}\ \mathit{Stmt},\ \texttt{eof}\,] \\ 4. & [\,\mathit{Stmt} \rightarrow \texttt{ifthen}\ \mathit{Stmt}\ \bullet\ \texttt{else}\ \mathit{Stmt},\ \texttt{else}\,] \end{array} \right\}$$

Each of those items generates an entry in the *Action* table. Unfortunately, they multiply define entries in the *Action* table. *Action*[8, `else`] is set to "*reduce 2*" by item 2 and to "*shift 10*" by item 4. Similarly, *Action*[8, `eof`] is set to "*reduce 2*" by item 1 and to "*shift 10*" by item 3. These multiple entries cause the table construction to fail.

The multiply-defined locations are the manifestation of an ambiguous grammar. It has two distinct rightmost derivations for some valid sentence, as evidenced by the fact that, in some valid contexts, the parser can either shift or reduce on the same lookahead symbol.

$$CC_0 = \left\{ \begin{array}{l} [\textit{Goal} \rightarrow \bullet\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}, \texttt{eof}] \\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{assign}, \texttt{eof}] \end{array} \right\}$$

$$CC_1 = \left\{ [\textit{Goal} \rightarrow \textit{Stmt}\ \bullet, \texttt{eof}] \right\}$$

$$CC_2 = \left\{ \begin{array}{l} [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}, \texttt{else}]\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \bullet\ \textit{Stmt}, \texttt{eof}] \\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{else}] \\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \bullet\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{assign}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{assign}, \texttt{else}] \end{array} \right\}$$

$$CC_3 = \left\{ [\textit{Stmt} \rightarrow \texttt{assign}\ \bullet, \texttt{eof}] \right\}$$

$$CC_4 = \left\{ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet, \texttt{eof}]\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet\ \texttt{else}\ \textit{Stmt}, \texttt{eof}] \right\}$$

$$CC_5 = \left\{ \begin{array}{l} [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}, \texttt{else}]\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \bullet\ \textit{Stmt}, \texttt{eof}] \\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \bullet\ \textit{Stmt}, \texttt{else}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{eof}] \\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{else}]\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \bullet\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{eof}] \\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \bullet\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{else}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{assign}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{assign}, \texttt{else}] \end{array} \right\}$$

$$CC_6 = \left\{ [\textit{Stmt} \rightarrow \texttt{assign}\ \bullet, \texttt{eof}]\ [\textit{Stmt} \rightarrow \texttt{assign}\ \bullet, \texttt{else}] \right\}$$

$$CC_7 = \left\{ \begin{array}{l} [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{eof}] \\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \bullet\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{assign}, \texttt{eof}] \end{array} \right\}$$

$$CC_8 = \left\{ \begin{array}{l} [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet, \texttt{eof}]\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet, \texttt{else}] \\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet\ \texttt{else}\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \bullet\ \texttt{else}\ \textit{Stmt}, \texttt{else}] \end{array} \right\}$$

$$CC_9 = \left\{ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}\ \bullet, \texttt{eof}] \right\}$$

$$CC_{10} = \left\{ \begin{array}{l} [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}, \texttt{else}] \\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}, \texttt{else}] \\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \bullet\ \textit{Stmt}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \bullet\ \textit{Stmt}, \texttt{else}] \\ [\textit{Stmt} \rightarrow \bullet\ \texttt{assign}, \texttt{eof}]\ [\textit{Stmt} \rightarrow \bullet\ \texttt{assign}, \texttt{else}] \end{array} \right\}$$

$$CC_{11} = \left\{ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}\ \bullet, \texttt{eof}]\ [\textit{Stmt} \rightarrow \texttt{ifthen}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}\ \bullet, \texttt{else}] \right\}$$

■ **FIGURE 3.29** The Canonical Collection of Sets of LR(1) Items for the `if–then–else` Grammar.

Shift-reduce error
An error that arises when the LR(1) table construction tries to define one entry in the *Action* table as both a *shift* and a *reduce*.

This particular kind of conflict is called a *shift-reduce conflict*. It arises because rule 2 in the grammar is a prefix of rule 3. The compiler writer needs to disambiguate the grammar, as shown in Section 3.2.3.

When the parser generator encounters such a conflict, the construction fails. The parser generator should report the problem back to the compiler writer. It is difficult to write that error message without reference to the LR(1) items.

Thus, most parser generators settle for a simple message such as "*shift–reduce error*" and provide a way for the user (presumably a compiler writer) to see the specific LR(1) items that were involved.

As an alternative, an LR(1) parser generator can simply resolve shift-reduce conflicts in favor of the shift action. This choice forces the parser to recognize the longer production. In the case of the `if–then–else` grammar, that decision binds each `else` to the innermost unmatched `ifthen`—precisely the rule that the disambiguated grammar from Section 3.2.3 enforces.

An ambiguous grammar can also give rise to a *reduce-reduce conflict*. Such a conflict occurs if the grammar contains two productions with the same right-hand side, $A \rightarrow \gamma\delta$ and $B \rightarrow \gamma\delta$. If some state CC_i contains both $[A \rightarrow \gamma\delta \bullet, \mathtt{a}]$ and $[B \rightarrow \gamma\delta \bullet, \mathtt{a}]$, then it will generate two conflicting reduce actions for *Action*$[i, \mathtt{a}]$. Again, this conflict reflects a fundamental ambiguity in the underlying grammar that the compiler writer must eliminate (see Section 3.5.3).

Reduce-reduce error
An ambiguity that causes the algorithm to assign two different reductions to a single *Action* table entry

Manually determining that a grammar has the LR(1) property is tedious and error-prone. LR(1) parser generators are widely available. Thus, the method of choice for determining if a given grammar has the LR(1) property is to invoke an LR(1) parser generator on it. If the process succeeds, the grammar has the LR(1) property.

SECTION REVIEW

LR(1) parsers are widely used in compilers built in both industry and academia. These parsers accept a large class of languages. They use time proportional to the size of the derivation that they construct. Tools that generate an LR(1) parser are widely available in many implementation languages.

The LR(1) table-construction algorithm is an elegant application of theory to practice. It systematically builds a model of the handle-recognizing DFA and encodes it into the *Action* and *Goto* tables. The construction requires painstaking attention to detail. It is precisely the kind of task that should be automated—parser generators are better at following these computations than are humans. That notwithstanding, a skilled compiler writer should understand the algorithms because they provide insight into how the parsers work, what kinds of errors the parser generator can encounter, how those errors arise, and how they can be fixed.

REVIEW QUESTIONS

1. Show the steps that the skeleton LR(1) parser, with the tables for the parentheses grammar, would take on the input string “(( ) ( )) ( ) .”
2. Build the canonical collection of sets of LR(1) items for the *SheepNoise* grammar, shown in the margin. Use the canonical collection to build *Action* and *Goto* tables. Show the resulting parser’s actions on the input “baa baa baa.”

SheepNoise → baa *SheepNoise*
| baa

3.5 PRACTICAL ISSUES

Despite the advent of high-quality parser generators, the compiler writer needs to understand a myriad of practical issues in order to build a quality parser. This section addresses several issues that arise in practice and have a significant impact on compiler usability.

3.5.1 Error Recovery

Programmers often compile code that contains syntax errors. In fact, compilers are widely accepted as the fastest way to discover such errors. In this application, the compiler must find as many syntax errors as possible in a single attempt at parsing the code. Thus, the compiler writer wants to build the parser so that it can move from an error state to one in which it can continue parsing.

The example parsers shown in this chapter all have the same behavior when they find a syntax error: they report the problem and halt. This behavior prevents the compiler from wasting more time on an incorrect program. However, it ensures that the compiler finds at most one syntax error per compilation. Finding all the syntax errors in a file of code with such a compiler could be a long and painful process.

A parser should find as many syntax errors as possible in each compilation. The parser needs a mechanism that lets it “recover” from an error—that is, move to a state where it can continue parsing. A common way to achieve this goal is to select one or more words that the parser can use to synchronize the input with its internal state. When the parser encounters an error, it discards input symbols until it finds a synchronizing word and then resets its internal state to one consistent with having recognized that word.

In an Algol-like language, with semicolons as statement separators, the semicolon is often used as a synchronizing word. When an error occurs, the parser calls the scanner repeatedly until it finds a semicolon. It then changes state to one that would have resulted from successful recognition of a complete statement, say the nonterminal *Stmt*, rather than an error.

In a recursive-descent parser, the code can simply discard words until it finds a semicolon. At that point, it can return control to the point where the routine that parses statements reports success. This may involve manipulating the runtime stack or using a nonlocal jump like C's `setjmp` and `longjmp`.

In an LR(1) parser, this kind of resynchronization is more complex. The parser discards input until it finds a semicolon. Next, it scans backward down the parse stack until it finds a state s such that *Goto*[s, *Stmt*] is a valid, nonerror entry. The first such state on the stack represents the statement that contains the error. The error recovery routine then discards entries on the stack down to that state, pushes the state given by *Goto*[s, *Stmt*] onto the stack, and resumes normal parsing.

In a table-driven parser, either LL(1) or LR(1), the compiler writer needs a way to tell the parser generator where to synchronize. This can be done using an error production—a production whose right-hand side includes a reserved word that indicates an error synchronization point and one or more synchronizing tokens. With such a construct, the parser generator can implement the desired behavior.

Of course, the compiler should not try to generate and optimize code for a syntactically invalid program. This requires simple handshaking between the error-recovery apparatus and the driver that invokes the compiler's various passes in order to halt after an unsuccessful parse.

3.5.2 **Unary Operators**

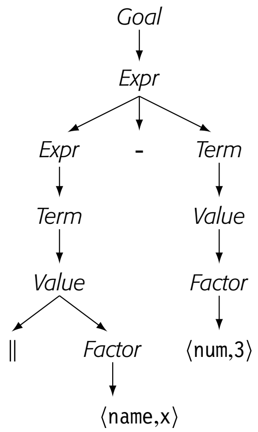
The classic expression grammar includes only binary operators. Algebraic notation, however, includes unary operators, such as unary minus and absolute value. Other unary operators arise in programming languages, including autoincrement, autodecrement, address-of, dereference, boolean complement, and typecasts. Adding such operators to the expression grammar requires some care.

Consider adding a unary absolute-value operator, ‖, to the classic expression grammar. Absolute value should have higher precedence than either × or ÷. However, it needs a lower precedence than *Factor* to force evaluation of parenthetic expressions before application of ‖. One way to write this grammar is shown in Fig. 3.30(a). With these additions, the grammar is still LR(1). It lets the programmer form the absolute value of any *Factor*.

Fig. 3.30(b) shows the parse tree for the string ‖ `x - 3`. It correctly shows that the code must evaluate ‖ `x` before performing the subtraction. The grammar

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Value*
5		|	*Term* ÷ *Value*
6		|	*Value*
7	*Value*	→	‖ *Factor*
8		|	*Factor*
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

(a) The Grammar



(b) Parse Tree for ‖ x - 3

■ **FIGURE 3.30** Adding Unary Absolute Value to the Classic Expression Grammar.

prevents the programmer from writing ‖ ‖ x, as that makes little mathematical sense. It does, however, allow ‖ (‖ x), which makes as little sense as ‖ ‖ x.

The inability to write ‖ ‖ x hardly limits the expressiveness of the language. With other unary operators, however, the issue seems more serious. For example, a C programmer might need to write `**p` to dereference a variable declared as `char **p;`. We can add a dereference production for *Value* as well: *Value* → * *Value*. The resulting grammar is still an LR(1) grammar, even if we replace the × operator in *Term* → *Term* × *Value* with *, overloading the operator "*" in the way that C does. This same approach works for unary minus.

3.5.3 Handling Context-Sensitive Ambiguity

Using one word to represent two distinct meanings can create a syntactic ambiguity. One example of this problem arose in several early programming languages, including FORTRAN, PL/I, and ADA. These languages used parentheses to enclose both the subscript expressions of an array reference and the argument list of a subroutine or function. Given a textual reference, such as `fee(i,j)`, the grammar has two derivations: one as a two-dimensional array and the other as a procedure call. Differentiating between these two cases requires knowledge of `fee`'s declared type—information that is not syntactically obvious. The scanner undoubtedly classifies `fee` as a `name` in either case.

Neither of these constructs appears in the classic expression grammar. We can add productions that derive them from *Factor*.

9	*Factor*	→	(*Expr*)
10		|	`num`
11		|	`name`
12		|	*Subscript*
13		|	*Call*
14	*Subscript*	→	`name` (*ArgList*)
15	*Call*	→	`name` (*ArgList*)

Since rules 14 and 15 expand to identical right-hand sides, this grammar is ambiguous. An LR(1) table builder will report a reduce-reduce conflict between these two productions.

In Section 3.3.1, we saw that using different brackets for *Subscript* and *Call* will also resolve the ambiguity.

Resolving this ambiguity requires extra-syntactic knowledge. In a top-down, recursive-descent parser, the compiler writer can combine the code for *Subscript* and *Call* and add the extra code required to check the `name`'s declared type. In a table-driven parser built with a parser generator, the solution must work within the framework provided by the tools.

Two different approaches have been used to solve this problem. The compiler writer can combine both the function invocation and the array reference into one production. This scheme defers the issue until later translation, when it can be resolved with type information derived from the declarations. The parser must construct a representation that preserves all the information needed by either resolution; some later step will rewrite the reference as either an array reference or a function invocation.

For a top-down parser, the compiler writer will also need to left-factor the occurrences of `name`.

Alternatively, the scanner can classify identifiers based on their declared types, rather than their microsyntax. This scheme requires hand-shaking between the scanner and the parser; it is not hard to arrange as long as the language has a define-before-use rule. The declaration is parsed before the use occurs. The parser can provide the scanner with access to it internal symbol table to resolve identifiers into distinct categories, such as `variable-name` and `function-name`. These categories then appear in the grammar as distinct nonterminal symbols. The relevant productions become:

Define-before-use
Many programming languages require that each name be declared before it is used in the code.

14	*Subscript*	→	`variable-name` (*ArgList*)
15	*Call*	→	`function-name` (*ArgList*)

Rewritten in this way, the grammar is unambiguous. The scanner returns a distinct syntactic category in each case and the parser can distinguish them.

Of course, the language designer can avoid this kind of ambiguity. Languages such as C and BCPL use different syntax for function calls and array element references, which eliminates the ambiguity.

SECTION REVIEW

Error localization, error messages, and error recovery are all critical to the compiler user's experience. The compiler writer needs to ensure that the compiler finds as many errors as it can and that it reports the causes of those errors as clearly as it can.

Unary operators and ambiguous constructs introduce complications into programming language grammars. The compiler writer can manage these issues by manipulating the grammar, by changing the parser–scanner interaction, or by some combination of both approaches.

REVIEW QUESTIONS

1. The programming language C uses square brackets to indicate an array subscript and parentheses to indicate a procedure or function call. How does this simplify the construction of a parser for C?
2. The grammar for unary absolute value introduced a new terminal symbol for the unary operator. Consider adding unary minus to the classic expression grammar. Does the fact that the same terminal symbol occurs in two distinct roles introduce complications? Justify your answer.

3.6 ADVANCED TOPICS

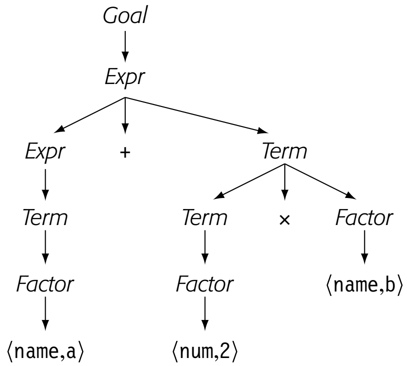
To build a satisfactory parser, the compiler writer must understand the basics of engineering both a grammar and a parser. Given a working parser, there are often ways of improving its performance. This section looks at two specific issues in parser construction. First, we examine transformations on the grammar that reduce the length of a derivation to produce a faster parse. These ideas apply to both top-down and bottom-up parsers. Second, we discuss transformations on the grammar and the *Action* and *Goto* tables that reduce table size. These techniques apply only to LR parsers.

3.6.1 Optimizing a Grammar

While syntax analysis no longer consumes a major share of compile time, the compiler should not waste undue time in parsing. The actual form of a

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	`num`
9		|	`name`

(a) The Classic Expression Grammar



(b) Parse Tree for a + 2 × b

FIGURE 3.31 The Classic Expression Grammar, Revisited.

grammar has a direct effect on the amount of work required to parse it. Both top-down and bottom-up parsers construct derivations. A top-down parser performs an expansion for every production in the derivation. A bottom-up parser performs a reduction for every production in the derivation. A grammar that produces shorter derivations takes less time to parse.

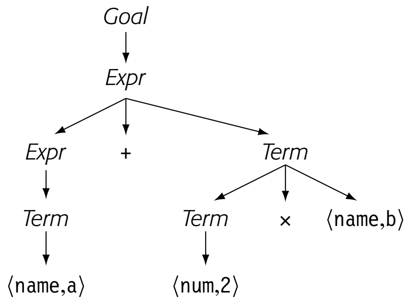
The compiler writer can often rewrite the grammar to reduce the parse tree height. This reduces the number of expansions in a top-down parser and the number of reductions in a bottom-up parser. Optimizing the grammar cannot change the parser's asymptotic behavior; after all, the parse tree must have a leaf node for each symbol in the input stream. Still, reducing the constants in heavily used portions of the grammar, such as the expression grammar, can make enough difference to justify the effort.

Consider, again, the classic left-recursive expression grammar, shown in Fig. 3.31(a). Its LR(1) tables appear in Fig. 3.33. This grammar produces rather large parse trees, even for simple expressions. For example, the parse tree for a + 2 × b, has 14 nodes, as shown in Fig. 3.31(b). Five of these nodes are leaves that we cannot eliminate; changing the grammar cannot shorten the input program. The number and placement of the interior nodes, however, depend completely on the derivation.

Any interior node that has only one child is a candidate for optimization. The sequence of nodes *Expr*→*Term*→*Factor*→⟨name, a⟩ uses four nodes for a single word in the input stream. We can eliminate at least one layer, the layer of *Factor* nodes, by folding the alternative expansions for *Factor* into *Term*, as shown in Fig. 3.32(a). It multiplies by three the number of alternatives for *Term*, but shrinks the parse tree by one layer, shown in Fig. 3.32(b).

4	*Term*	→	*Term* × (*Expr*)
5		|	*Term* × `name`
6		|	*Term* × `num`
7		|	*Term* ÷ (*Expr*)
8		|	*Term* ÷ `name`
9		|	*Term* ÷ `num`
10		|	(*Expr*)
11		|	`num`
12		|	`name`

(a) New Productions for *Term*



(b) Parse Tree for a + 2 × b

■ **FIGURE 3.32** Replacement Productions for *Term*.

In an LR(1) parser, this change eliminates three of nine reduce actions, and leaves the five shifts intact. In a top-down, recursive-descent parser for an equivalent predictive grammar, it would eliminate 3 of 14 procedure calls.

In general, any production that has a right-hand side with one symbol can be folded away. We call such productions *useless*. Sometimes, useless productions serve a purpose—making the grammar more compact and, perhaps, more readable, or forcing the derivation to assume a particular shape. (Recall that the simplest of our expression grammars accepts a + 2 × b but does not encode any notion of precedence into the parse tree.)

In Section 5.3, we will discuss methods to tie computations to particular points in the parse. In that context, a production that is useless in the syntax may play a critical role in the translation by creating a point in the derivation where a particular action can be performed.

Folding away useless productions has its costs. In an LR(1) parser, it can make the tables larger. In our example, eliminating *Factor* removes one column from the *Goto* table, but the extra productions for *Term* increase the size of *CC* from 32 sets to 46 sets. Thus, the tables have one fewer column, but an extra 14 rows. The resulting parser performs fewer reductions (and runs faster), but has larger tables.

In a hand-coded, recursive-descent parser, the larger grammar may increase the number of alternatives that must be compared before expanding some left-hand side. The compiler writer can sometimes compensate for the increased cost by combining cases. For example, in our recursive-descent parser in Fig. 3.10, the code in routine `ExprPrime` for the words + and - was combined because the parser takes the same actions in either case.

3.6.2 Reducing the Size of LR(1) Tables

Unfortunately, the LR(1) tables generated for even small grammars can be large. Fig. 3.33 on page 150 shows the *Action* and *Goto* tables for the classic expression grammar. Many techniques exist for reducing LR(1) table sizes. This section describes three such approaches.

Shrinking the Grammar

The compiler writer can often recode the grammar to reduce the number of productions. Fewer productions usually lead to smaller tables. For example, in the classic expression grammar, the distinction between a number and an identifier is irrelevant to the productions for *Goal*, *Expr*, *Term*, and *Factor*. Replacing the two productions *Factor* → `num` and *Factor* → `name` with a single production *Factor* → `val` shrinks the grammar by a production. In the *Action* table, each terminal symbol has its own column. Folding `num` and `name` into a single symbol, `val`, removes one of those columns. Of course, the scanner must then return the same syntactic category, or word, for both `num` and `name`.

Similar arguments can be made for combining × and ÷ into a terminal `muldiv`, and for combining + and - into a terminal `addsub`. Each of these replacements removes a terminal symbol and a production. These three changes produce the reduced expression grammar shown in Fig. 3.34(a). This grammar produces a smaller $\mathcal{CC}$, removing rows from the table. Because the grammar has fewer terminal symbols, the table has fewer columns as well.

Combining + and - into one category has the same effect as combining cases in the top-down, recursive-descent parser, discussed in Section 3.6.1.

The resulting *Action* and *Goto* tables are shown in Fig. 3.34(b). The *Action* table contains 132 entries and the *Goto* table contains 66 entries, for a total of 198 entries. This compares favorably with the tables for the original grammar, with their 384 entries. Changing the grammar produced a 48 percent reduction in table size.

Other considerations may limit the compiler writer's ability to combine productions. In particular, two similar productions may need different syntax-driven translation schemes, which would make combining them impractical (see Section 5.3).

Combining Rows or Columns

Shrinking the grammar can produce a significant reduction in the size of the *Action* and *Goto* tables. The resulting tables may still present opportunities for improvement. Consider the tables in Fig. 3.34(b), for example. The *Action* table rows for states 0, 6, and 7 are identical, as are the rows for states 4, 11, 15, and 17. The table generator can combine rows, or columns, that are

	Action Table									Goto Table		
State	**eof**	**+**	**-**	**×**	**÷**	**(**	**)**	**num**	**name**	***Expr***	***Term***	***Factor***
0						s 4		s 5	s 6	1	2	3
1	acc	s 7	s 8									
2	r 4	r 4	r 4	s 9	s 10							
3	r 7	r 7	r 7	r 7	r 7							
4						s 14		s 15	s 16	11	12	13
5	r 9	r 9	r 9	r 9	r 9							
6	r 10	r 10	r 10	r 10	r 10							
7						s 4		s 5	s 6		17	3
8						s 4		s 5	s 6		18	3
9						s 4		s 5	s 6			19
10						s 4		s 5	s 6			20
11		s 21	s 22				s 23					
12		r 4	r 4	s 24	s 25		r 4					
13		r 7	r 7	r 7	r 7		r 7					
14						s 14		s 15	s 16	26	12	13
15		r 9	r 9	r 9	r 9		r 9					
16		r 10	r 10	r 10	r 10		r 10					
17	r 2	r 2	r 2	s 9	s 10							
18	r 3	r 3	r 3	s 9	s 10							
19	r 5	r 5	r 5	r 5	r 5							
20	r 6	r 6	r 6	r 6	r 6							
21						s 14		s 15	s 16		27	13
22						s 14		s 15	s 16		28	13
23	r 8	r 8	r 8	r 8	r 8							
24						s 14		s 15	s 16			29
25						s 14		s 15	s 16			30
26		s 21	s 22				s 31					
27		r 2	r 2	s 24	s 25		r 2					
28		r 3	r 3	s 24	s 25		r 3					
29		r 5	r 5	r 5	r 5		r 5					
30		r 6	r 6	r 6	r 6		r 6					
31		r 8	r 8	r 8	r 8		r 8					

■ **FIGURE 3.33** *Action* and *Goto* Tables for the Classic Expression Grammar.

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* `addsub` *Term*
3		|	*Term*
4	*Term*	→	*Term* `muldiv` *Factor*
5		|	*Factor*
6	*Factor*	→	(*Expr*)
7		|	`val`

(a) The Reduced Expression Grammar

	Action Table						Goto Table		
State	`eof`	`addsub`	`muldiv`	`(`	`)`	`val`	*Expr*	*Term*	*Factor*
0				s 4		s 5	1	2	3
1	acc	s 6							
2	r 3	r 3	s 7						
3	r 5	r 5	r 5						
4				s 11		s 12	8	9	10
5	r 7	r 7	r 7						
6				s 4		s 5		13	3
7				s 4		s 5			14
8		s 15			s 16				
9		r 3	s 17		r 3				
10		r 5	r 5		r 5				
11				s 11		s 12	18	9	10
12		r 7	r 7		r 7				
13	r 2	r 2	s 7						
14	r 4	r 4	r 4						
15				s 11		s 12		19	10
16	r 6	r 6	r 6						
17				s 11		s 12			20
18		s 15			s 21				
19		r 2	s 17		r 2				
20		r 4	r 4		r 4				
21		r 6	r 6		r 6				

(b) *Action* and *Goto* Tables for the Reduced Expression Grammar

■ **FIGURE 3.34** The Reduced Expression Grammar and Its Tables.

Together, the techniques reduced the table from 288 to 102 entries, or 65 percent.

identical and remap the states accordingly. Combining the identical rows in the two sets above would further reduce the *Action* table in Fig. 3.34(b) from 132 entries to 102 entries, an additional 23 percent reduction.

To use the resulting table, the skeleton parser would need a mapping from a parser state to a row index in the *Action* table. The table generator can combine identical columns in the analogous way. A separate inspection of the *Goto* table will yield a different set of state combinations—in particular, all of the rows containing only zeros should condense to a single row.

In some cases, the table generator can prove that two rows or two columns differ only in cases where one of the two has an "error" entry (denoted by a blank in our figures). In Fig. 3.33, the columns for `eof` and for `num` differ only where one or the other has a blank. Combining such columns produces the same behavior on correct inputs. It does change the parser's behavior on erroneous inputs and may impede the parser's ability to provide accurate and helpful error messages.

The mechanics of combining rows and columns is similar to the technique discussed for shrinking the transition-function table of a DFA, discussed in Section 2.5.4. These techniques produce a direct reduction in table size. If this space reduction adds an extra indirection to every table access, the cost of those memory operations must trade off directly against the savings in memory. The table generator could also use other techniques to represent sparse matrices—again, the implementor must consider the tradeoff of memory size against any increase in access costs.

Directly Encoding the Table

As a final improvement, the parser generator can abandon the table-driven skeleton parser in favor of a direct-coded implementation. Each state becomes a small case statement or a collection of `if–then–else` statements that tests the type of the next symbol and either shifts, reduces, accepts, or reports an error. The *Action* and *Goto* tables are replaced with code. (We saw a similar transformation for scanners in Section 2.5.2.)

The resulting parser avoids directly representing all of the "don't care" states in the *Action* and *Goto* tables, shown as blanks in the figures. This space savings may be offset by larger code size, since each state now requires code. The new parser, however, has no parse table, performs no table lookups, and lacks the outer loop found in the skeleton parser. While its structure makes it almost unreadable by humans, it should execute more quickly than the corresponding table-driven parser. With appropriate code-layout techniques, the resulting parser can exhibit strong locality in both cache and virtual memory. For example, the compiler writer should place routines for the expression grammar together on a single page in virtual memory, so that they cannot conflict with one another in the code cache.

Using Other Construction Algorithms

Several other algorithms to construct LR-style parsers exist. Among these techniques are the SLR(1) construction, for simple LR(1), and the LALR(1) construction, for lookahead LR(1). Both of these constructions produce smaller tables than the canonical LR(1) algorithm.

The SLR(1) algorithm accepts a smaller class of grammars than the canonical LR(1) construction. The SLR(1) grammars are restricted so that the table-filling algorithm can distinguish between shift entries and reduce entries using the FOLLOW sets. This restriction eliminates the need for lookahead symbols in the items. The resulting canonical collection of sets of items has fewer states, so the table has fewer rows. This technique accepts many grammars of practical interest.

The LALR(1) algorithm relies on the observation that the core items in the set representing a state are critical and that the remaining items can be added to a set by computing its *closure*. The LALR(1) table construction uses the core items to compute the canonical collection and computes their closure after reaching the fixed point. It produces a canonical collection similar in size to the one produced by the SLR(1) construction. The details differ, but the table sizes are similar.

Recall that the *core* items of a set are those produced by the *goto* functions. The items added by *closure* are not core items.

The canonical LR(1) construction presented earlier in the chapter is the most general of these table-construction algorithms. It produces the largest tables, but accepts the largest class of grammars. With appropriate table reduction techniques, the LR(1) tables can approximate the size of those produced by the more limited techniques.

The SLR(1) construction accepts a smaller class of grammars than the LALR(1) construction. In turn, the LALR(1) construction accepts a smaller class of grammars than does the canonical LR(1) construction. However, in a mildly counterintuitive result, any language that has an LR(1) grammar also has both an SLR(1) grammar and an LALR(1) grammar. The grammars for these more restrictive construction algorithms must be shaped in ways that let the algorithms differentiate between shift actions and reduce actions.

3.7 **SUMMARY AND PERSPECTIVE**

Almost every compiler contains a parser. For many years, parsing was a subject of intense research interest. This led to the development of many different techniques for building efficient parsers. The LR(1) family of grammars includes all of the CFGs that can be parsed in a deterministic fashion. The tools produce efficient parsers with provably strong error-detection properties. This combination of features, coupled with the widespread availability of parser generators for LR(1), SLR(1) and LALR(1) grammars, has decreased interest in other bottom-up parsing techniques such as operator precedence.

Top-down, recursive-descent parsers have their own set of advantages. They are, arguably, the easiest hand-coded parsers to construct. They provide excellent opportunities to detect and repair syntax errors. They are efficient; in fact, a well-constructed top-down, recursive-descent parser can be faster than a table-driven LR(1) parser. (The direct encoding scheme for LR(1) may overcome this speed advantage.) In a top-down, recursive-descent parser, the compiler writer can more easily finesse ambiguities in the source language that might trouble an LR(1) parser—such as a language in which keyword names can appear as identifiers. A compiler writer who wants to construct a hand-coded parser, for whatever reason, is well advised to use the top-down, recursive-descent method.

For an example of a language that has an LR(1) grammar and no equivalent LL(1) grammar, see Exercise 3.10.

In choosing between LR(1) and LL(1) grammars, the choice often depends on the availability of tools. In practice, few, if any, programming-language constructs fall in the gap between LR(1) grammars and LL(1) grammars. Thus, starting with an available parser generator is always better than implementing a parser generator from scratch.

More general parsing algorithms are available. In practice, however, the restrictions that LR(1) and LL(1) impose on CFGs seem to allow language designers to express most features found in actual programming languages.

CHAPTER NOTES

The earliest compilers used hand-coded parsers [28,238,324]. The rich syntax of ALGOL-60 challenged early compiler writers. They tried a variety of schemes to parse the language; Randell and Russell provide an overview of the methods used in ALGOL-60 compilers [303].

Irons was one of the first to separate the notion of syntax from translation [214]. Lucas appears to have introduced the notion of recursive-descent parsing [263]. Conway applies similar ideas to an efficient single-pass compiler for COBOL [103].

The ideas behind LL and LR parsing were developed in the 1960s. Lewis and Stearns introduced LL(k) grammars [253], while Rosenkrantz and Stearns described the properties of LL(k) grammars in more depth [313].

Foster developed an algorithm to transform some grammars into LL(1) form [161]. Wood formalized the notion of left-factoring a grammar and explored the theoretical issues involved in transforming a grammar to LL(1) form [365–367].

Knuth laid out the theory behind LR(1) parsing [239]. DeRemer and others developed the SLR(1) and LALR(1) table-construction algorithms, which made the use of LR parser generators practical on the limited-memory computers of the day [132,133]. Waite and Goos describe a technique for automatically eliminating useless productions during the LR(1) table-construction algorithm [352]. Penello suggested direct encoding of the tables into executable code [291]. Aho and Ullman [9] is a definitive reference on both LL and LR parsing. Bill Waite provided the example grammar in Exercise 3.10.

Several algorithms for parsing arbitrary CFGs appeared in the 1960s and 1970s. Cocke and Schwartz [98], Earley [146], Kasami [223], and Younger [370] all developed algorithms of similar computational complexity. Earley's algorithm deserves note because of its similarity to the LR(1) table-construction algorithm. His algorithm derives the set of possible parse states at parse time, rather than at runtime, where the LR(1) techniques precompute them in a parser generator. From a high-level view, the LR(1) algorithms might appear as a natural optimization of Earley's algorithm.

EXERCISES

Section 3.2

1. Write a context-free grammar for the syntax of regular expressions.
2. When asked to define an *unambiguous context-free grammar* on an exam, two students gave different answers. The first defined it as "a grammar where each sentence has a unique parse tree by leftmost derivation." The second defined it as "a grammar where each sentence has a unique parse tree by any derivation." Which one is correct?
3. Consider a robot arm that accepts two commands: ▽ puts an apple in the bag and △ takes an apple out of the bag. Assume the robot arm starts with an empty bag.

   A valid command sequence for the robot arm has no prefix that contains more △ commands than ▽ commands. Thus, ▽▽△△ and ▽△▽ are valid command sequences, but ▽△△▽ and ▽△▽△△ are not.

   Write a grammar that represents all the valid command sequences for the robot arm.

Section 3.3

4. The following grammar is not suitable for a top-down predictive parser. Identify the problem and correct it by rewriting the grammar. Show that your new grammar satisfies the LL(1) condition.

1	*L*	→	*R* a
2		|	*Q* ba
3	*R*	→	aba
4		|	caba
5		|	*R* bc
6	*Q*	→	bbc
7		|	bc

5. Grammars that can be parsed top-down, in a linear scan from left to right, with a k word lookahead are called LL(k) grammars. In the text, the LL(1) condition is described in terms of FIRST sets. How would you define the FIRST sets necessary to describe an LL(k) condition?

6. Suppose an elevator is controlled by two commands: ↑ to move the elevator up one floor and ↓ to move the elevator down one floor. Assume that the building is arbitrarily tall and that the elevator starts at floor x.

   Write an LL(1) grammar that generates arbitrary command sequences that (1) never cause the elevator to go below floor x and (2) always return the elevator to floor x at the end of the sequence. For example, ↑↑↓↓ and ↑↓↑↓ are valid command sequences, but ↑↓↓↑ and ↑↓↓ are not. The null sequence is valid. Prove that your grammar is LL(1).

Section 3.4

7. Top-down and bottom-up parsers build parse trees in different orders. Write a pair of programs, `TopDown` and `BottomUp`, that take a parse tree and print out the nodes in order of construction. `TopDown` should display the order for a top-down parser, while `BottomUp` should show the order for a bottom-up parser.

8. The *ClockNoise* language (*CN*) is represented by the following grammar:

1	*Goal*	→	*ClockNoise*
2	*ClockNoise*	→	*ClockNoise* tick tock
3		|	tick tock

   a. What are the LR(1) items of *CN*?

   b. What are the FIRST sets of *CN*?

   c. Construct the Canonical Collection of Sets of LR(1) Items for *CN*.

   d. Derive the *Action* and *Goto* tables.

9. Consider the following grammar:

1	*Start*	→	*S*
2	*S*	→	*A* a
3	*A*	→	*B C*
4		|	*B C* f
5	*B*	→	b
6	*C*	→	c

   a. Construct the canonical collection of sets of LR(1) items for this grammar.

   b. Derive the *Action* and *Goto* tables.

   c. Is the grammar LR(1)?

10. The following grammar has no known LL(1) equivalent:

1	*Start*	→	*A*
2		|	*B*
3	*A*	→	(*A*)
4		|	a
5	*B*	→	(*B*]
6		|	b

Show that the grammar is LR(1).

Section 3.6

11. Write a grammar for expressions that includes binary operators (+ and ×), unary minus (-), autoincrement (++), and autodecrement (--) with their customary precedence. Assume that repeated unary minuses are not allowed, but that repeated autoincrement and autodecrement operators are allowed.

Section 3.7

12. Consider the task of building a parser for the programming language SCHEME. Contrast the effort required for a top-down recursive-descent parser with that needed for a table-driven LR(1) parser. (Assume that you already have an LR(1) table generator.)

This page intentionally left blank

Chapter 4

Intermediate Representations

ABSTRACT
The central data structure in a compiler is its representation of the program being compiled. Most passes in the compiler read and manipulate this intermediate representation or IR. Thus, decisions about what to represent and how to represent it play a crucial role in both the cost of compilation and its effectiveness. This chapter presents a survey of IRs that compilers use, including graphical IRs, linear IRs, and hybrids of these two forms, along with the ancillary data structures that the compiler maintains, typified by its symbol tables.

KEYWORDS
Intermediate Representation, Graphical IR, Linear IR, SSA Form, Symbol Table, Memory Model, Storage Layout

4.1 INTRODUCTION

Compilers are typically organized as a series of passes. As the compiler derives knowledge about the code it translates, it must record that knowledge and convey it to subsequent passes. Thus, the compiler needs a representation for all of the facts that it derives about the program. We call this collection of data structures an intermediate representation (IR). A compiler may have one IR, or it may have a series of IRs that it uses as it translates from the source code into the target language. The compiler relies on the IR to represent the program; it does not refer back to the source text. The properties of the IR have a direct effect on what the compiler can and cannot do to the code.

Use of an IR lets the compiler make multiple passes over the code. The compiler can generate more efficient code for the input program if it can gather information in one pass and use it in another. However, this capability imposes a requirement: the IR must be able to represent the derived information. Thus, compilers also build a variety of ancillary data structures to represent derived information and provide efficient access to it. These structures also form part of the IR.

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00010-3

Almost every phase of the compiler manipulates the program in its IR form. Thus, the properties of the IR, such as the methods for reading and writing specific fields, for finding specific facts, and for navigating around the program, have a direct impact on the ease of writing the individual passes and on the cost of executing those passes.

Conceptual Roadmap

This chapter focuses on the issues that surround the design and use of IRs in compilation. Some compilers use trees and graphs to represent the program being compiled. For example, parse trees easily capture the derivations built by a parser and Lisp's S-expressions are, themselves, simple graphs. Because most processors rely on a linear assembly language, compilers often use linear IRs that resemble assembly code. Such a linear IR can expose properties of the target machine's native code that provide opportunities to the compiler.

As the compiler builds up the IR form of the program, it discovers and derives information that may not fit easily into a tree, graph, or linear IR. It must understand the name space of the program and build ancillary structures to record that derived knowledge. It must create a plan for the layout of storage so that the compiled code can store values into memory and retrieve them as needed. Finally, it needs efficient access, by name, to all of its derived information. To accommodate these needs, compilers build a set of ancillary structures that coexist with the tree, graph, or linear IR and form a critical part of the compiler's knowledge base about the program.

Overview

Modern multipass compilers use some form of IR to model the code being analyzed, translated, and optimized. Most passes in the compiler consume IR; the stream of categorized words produced by the scanner can be viewed as an IR designed to communicate between the scanner and the parser. Most passes in the compiler produce IR; passes in the code generator can be exceptions. Many modern compilers use multiple IRs during the course of a single compilation. In a pass-structured compiler, the IR serves as the primary representation of the code.

A compiler's IR must be expressive enough to record all of the useful facts that the compiler might need to transmit between passes. Source code is insufficient for this purpose; the compiler derives many facts that have no representation in source code. Examples include the addresses of variables or the register number in which a given parameter is passed. To record all of the details that the compiler must encode, most compiler writers augment the IR with tables and sets that record additional information. These structures form an integral part of the IR.

Selecting an appropriate IR for a compiler project requires an understanding of the source language, the target machine, the goals for the compiler, and the properties of the applications that the compiler will translate. For example, a source-to-source translator might use a parse tree that closely resembles the source code, while a compiler that produces assembly code for a microcontroller might obtain better results with a low-level assembly-like IR. Similarly, a compiler for C might need annotations about pointer values that are irrelevant in a LISP compiler. Compilers for JAVA or C++ record facts about the class hierarchy that have no counterpart in a C compiler.

Implementing an IR forces the compiler writer to focus on practical issues. The IR is the compiler's central data structure. The compiler needs inexpensive ways to perform the operations that it does frequently. It needs concise ways to express the full range of constructs that might arise during compilation. Finally, the compiler writer needs mechanisms that let humans examine the IR program easily and directly; self-interest should ensure that compiler writers pay heed to this last point.

Common operations should be inexpensive. Uncommon operations should be doable at a reasonable cost.

For example, ILOC's ⇒ symbol has one purpose: to improve readability.

The remainder of this chapter explores the issues that arise in the design and use of IRs. Section 4.2 provides a taxonomy of IRs and their properties. Section 4.3 describes several IRs based on trees and graphs, while Section 4.4 presents several common linear forms of IRs. Section 4.5 provides a high-level overview of symbol tables and their uses; Appendix B.4 delves into some low-level hash-table implementation issues. The final two sections, 4.6 and 4.7, explore issues that arise from the way that the compiler names values and the rules that the compiler applies to place values in memory.

A Few Words About Time

Intermediate representations are, almost entirely, a compile-time construct. Thus, the compiler writer has control over the IR design choices, which she makes at design time. The IR itself is instantiated, used, and discarded at compile time.

Some of the ancillary information generated as part of the IR, such as symbol tables and storage maps, is preserved for later tools, such as the debugger. Those use cases, however, do not affect the design and implementation of the IR because that information must be translated into some standard form dictated by the tools.

4.2 AN IR TAXONOMY

Compilers have used many kinds of IR. We will organize our discussion of IRs along three axes: structural organization, level of abstraction, and mode of use. In general, these three attributes are independent; most combinations of organization, abstraction, and naming have been used in some compiler.

Structural Organization

Broadly speaking, IRs fall into three classes:

Graphical IRs encode the compiler's knowledge in a graph. Algorithms then operate over nodes and edges. The parse trees used to depict derivations in Chapter 3 are an instance of a graphical IR, as are the trees shown in panels (a) and (c) of Fig. 4.1.

Linear IRs resemble pseudocode for some abstract machine. The algorithms iterate over simple, linear sequences of operations. The ILOC code used in this book is a form of linear IR, as are the representations shown in panels (b) and (d) of Fig. 4.1.

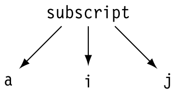
Hybrid IRs combine elements of both graphical and linear IRs, to capture their strengths and avoid their weaknesses. A typical control-flow graph (CFG) uses a linear IR to represent blocks of code and a graph to represent the flow of control among those blocks.

Compiler writers use the acronym CFG for both *context-free grammar* and *control-flow graph*. The difference should be clear from context.

The structural organization of an IR has a strong impact on how the compiler writer thinks about analysis, optimization, and code generation. For example, tree-structured IRs lead naturally to passes organized as some form of treewalk. Similarly, linear IRs lead naturally to passes that iterate over the operations in order.

Level of Abstraction

The compiler writer must also choose the level of detail that the IR will expose: its level of abstraction. The IR can range from a near-source form in which a couple of nodes represent an array access or a procedure call to a low-level form in which multiple IR operations must be combined to form a single target-machine operation. To illustrate the possibilities, the drawing in the margin shows a reference to a[i,j] represented in a source-level tree. Below it, the same reference is shown in ILOC. In both cases, a is a 10 × 10 array of 4-byte elements.

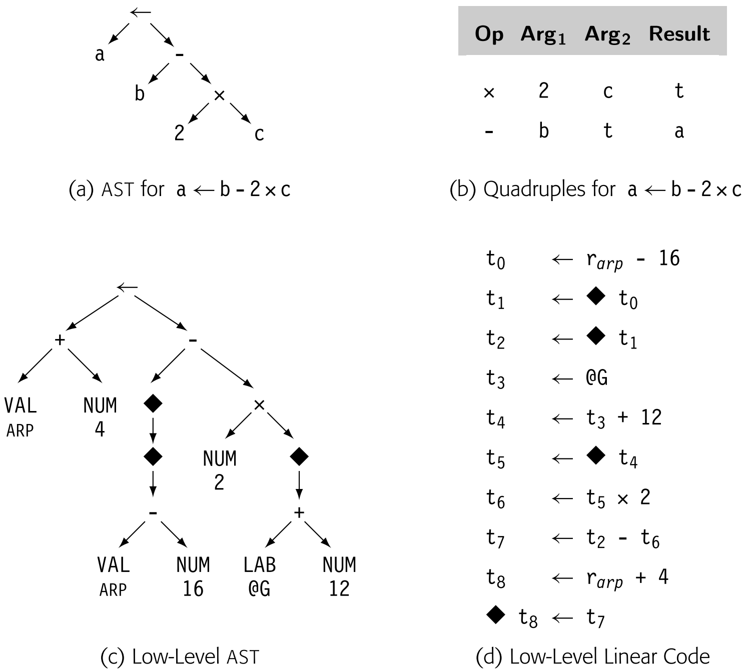


Source-Level Tree

```
subI    r_i, 1    ⇒ r_1
multI   r_1, 10   ⇒ r_2
subI    r_j, 1    ⇒ r_3
add     r_2, r_3  ⇒ r_4
multI   r_4, 4    ⇒ r_5
loadI   @a        ⇒ r_6
add     r_5, r_6  ⇒ r_7
load    r_7       ⇒ r_aij
```

ILOC Code

In the source-level tree, the compiler can easily recognize the computation as an array reference, whereas the ILOC code obscures that fact fairly well. In a compiler that tries to determine when two different references can touch



(a) AST for a ← b - 2 × c

Op	Arg$_1$	Arg$_2$	Result
×	2	c	t
-	b	t	a

(b) Quadruples for a ← b - 2 × c

(c) Low-Level AST

```
t0    ← rarp - 16
t1    ← ◆ t0
t2    ← ◆ t1
t3    ← @G
t4    ← t3 + 12
t5    ← ◆ t4
t6    ← t5 × 2
t7    ← t2 - t6
t8    ← rarp + 4
◆ t8  ← t7
```

(d) Low-Level Linear Code

■ **FIGURE 4.1** Different Representations for a ← b - 2 x c.

the same memory location, the source-level tree makes it easy to find and compare references. By contrast, the ILOC code makes those tasks hard. On the other hand, if the goal is to optimize the final code generated for the array access, the ILOC code lets the compiler optimize details that remain implicit in the source-level tree. For this purpose, a low-level IR may prove better.

Level of abstraction is independent of structure. Fig. 4.1 shows four different representations for the statement a ← b - 2 x c. Panels (a) and (c) show abstract syntax trees (ASTs) at both a near-source level and a near-machine level of abstraction. Panels (b) and (d) show corresponding linear representations.

The low-level AST in panel (c) uses nodes that represent assembly-level concepts. A VAL node represents a value already in a register. A NUM node represents a known constant that can fit in an operation's immediate field. A LAB node represents an assembly-level label. The dereference operator, ◆, treats the value as an address and represents a memory reference. This particular AST will reappear in Chapter 11.

The translation of ◆ in the low-level linear code depends on context. To the left of a ← operator, it represents a `store`. To the right, it represents a `load`.

Level of abstraction matters because the compiler can, in general, only optimize details that the IR exposes. Facts that are implicit in the IR are hard to change because the compiler treats implicit knowledge in uniform ways, which mitigates against context-specific customization. For example, to optimize the code for an array reference, the compiler must rewrite the IR for the reference. If the details of that reference are implicit, the compiler cannot change them.

Mode of Use

The third axis relates to the way that the compiler uses an IR.

Definitive IR A *definitive* IR is the primary representation for the code being compiled. The compiler does not refer back to the source code; instead, it analyzes, transforms, and translates one or more (successive) IR versions of the code. These IRs are definitive IRs.

Derivative IR A *derivative* IR is one that the compiler builds for a specific, temporary purpose. The derivative IR may augment the definitive IR, as with a dependence graph for instruction scheduling (see Chapter 12). The compiler may translate the code into and out of the derivative IR to enable a specific optimization.

In general, if an IR is transmitted from one pass to another, it should be considered definitive. If the IR is built within a pass for a specific purpose and then discarded, it is derivative.

Naming

The compiler writer must also select a name space for the IR. This decision will determine which values in the program are exposed to optimization. As it translates the source code, the compiler must choose names and storage locations for myriad distinct values.

Fig. 4.1 makes this concrete. In the ASTs, the names are implicit; the compiler can refer to any subtree in the AST by the node that roots the subtree. Thus, the tree in panel (c) names many values that cannot be named in panel (a), because of its lower level of abstraction. The same effect occurs in the linear codes. The code in panel (b) creates just two values that other operations can use while the code in panel (d) creates nine.

The naming scheme has a strong effect on how optimization can improve the code. In panel (d), t_0 is the runtime address of b, t_4 is the runtime address of c, and t_8 is the runtime address of a. If nearby code references any of these locations, optimization should recognize the identical references and reuse the computed values (see Section 8.4.1). If the compiler reused the name t_0 for another value, the computed address of b would be lost, because it could not be named.

REPRESENTING STRINGS

The scanner classifies words in the input program into a small set of categories. From a functional perspective, each word in the input stream becomes a pair ⟨ *lexeme, category* ⟩, where *lexeme* is the word's text and *category* is its syntactic category.

For some categories, having both *lexeme* and *category* is redundant. The categories `+`, `×`, and `for` have only one lexeme. For others, such as identifiers, numbers, and character strings, distinct words have distinct lexemes. For these categories, the compiler will need to represent and compare the lexemes.

Character strings are one of the least efficient ways that the compiler can represent a name. The character string's size is proportional to its length. To compare two strings takes, in general, time proportional to their length. A compiler can do better.

The compiler should, instead, map the names used in the original code into a compact set of integers. This approach saves space; integers are denser than character strings. This approach saves time; comparisons for equality take $\mathbf{O}(1)$ time.

To accomplish this mapping, the compiler writer can have the scanner create a hash table (see Appendix B.4) to hold all the distinct strings used in the input program. Then the compiler can use either the string's index in this "string table" or a pointer to its record in the string table as a proxy for the string. Information derived from the string, such as the length of a string constant or the value and type of a numerical constant, can be computed once and referenced quickly through the table.

Using too few names can undermine optimization. Using too many can bloat some of the compile-time data structures and increase compile time without benefit. Section 4.6 delves into these issues.

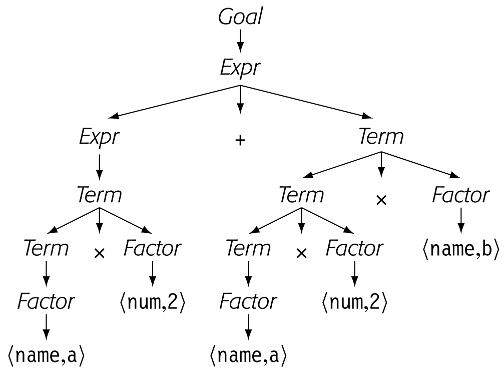
Practical Considerations

As a practical matter, the costs of generating and manipulating an IR should concern the compiler writer, since they directly affect a compiler's speed. The data-space requirements of different IRs vary over a wide range. Since the compiler typically touches all of the space that it allocates, data space usually has a direct relationship to running time.

Last, and certainly not least, the compiler writer should consider the expressiveness of the IR—its ability to accommodate all the facts that the compiler needs to record. The IR for a procedure might include the code that defines it, the results of static analysis, profile data from previous executions, and

Goal	→	*Expr*
Expr	→	*Expr* + *Term*
	|	*Expr* - *Term*
	|	*Term*
Term	→	*Term* × *Factor*
	|	*Term* ÷ *Factor*
	|	*Factor*
Factor	→	(*Expr*)
	|	`num`
	|	`name`

(a) Classic Expression Grammar



(b) Parse Tree for a × 2 + a × 2 × b

■ **FIGURE 4.2** Parse Tree for a × 2 + a × 2 × b.

maps to let the debugger understand the code and its data. All of these facts should be expressed in a way that makes clear their relationship to specific points in the IR.

4.3 GRAPHICAL IRS

Many compilers use IRs that represent the underlying code as a graph. While all the graphical IRs consist of nodes and edges, they differ in their level of abstraction, in the relationship between the graph and the underlying code, and in the structure of the graph.

4.3.1 Syntax-Related Trees

Parse trees, ASTs, and directed acyclic graphs (DAGs) are all graphs used to represent code. These tree-like IRs have a structure that corresponds to the syntax of the source code.

Parse Trees

As we saw in Section 3.2, the *parse tree* is a graphical representation for the derivation, or parse, of the input program. Fig. 4.2 shows the classic expression grammar alongside a parse tree for a × 2 + a × 2 × b. The parse tree is large relative to the source text because it represents the complete derivation, with a node for each grammar symbol in the derivation. Since the compiler must allocate memory for each node and each edge, and it must traverse all those nodes and edges during compilation, it is worth considering ways to shrink this parse tree.

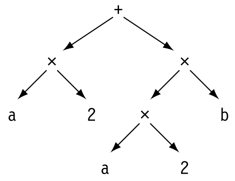
Minor transformations on the grammar, as described in Section 3.6.1, can eliminate some of the steps in the derivation and their corresponding parse-tree nodes. A more effective way to shrink the parse tree is to abstract away those nodes that serve no real purpose in the rest of the compiler. This approach leads to a simplified version of the parse tree, called an *abstract syntax tree*, discussed below.

Mode of Use: Parse trees are used primarily in discussions of parsing, and in attribute-grammar systems, where they are the definitive IR. In most other applications in which a source-level tree is needed, compiler writers tend to use one of the more concise alternatives, such as an AST or a DAG.

Abstract Syntax Trees

The *abstract syntax tree* (AST) retains the structure and meaning of the parse tree but eliminates extraneous nodes. It eliminates the nodes for nonterminal symbols that encode the details of the derivation. An AST for a × 2 + a × 2 × b is shown in the margin.

Abstract syntax tree
a contraction of the parse tree that omits nodes for most nonterminals



AST for a × 2 + a × 2 × b

Mode of Use: ASTs have been used as the definitive IR in many practical compiler systems. The level of abstraction that those systems need varies widely.

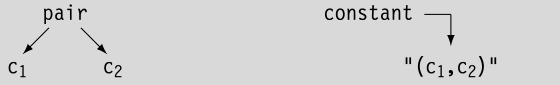
- Source-to-source systems, including syntax-directed editors, code-refactoring tools, and automatic parallelization systems, often use an AST with near-source abstractions. The structure of a near-source AST reflects the structure of the input program.
- Compilers that generate assembly code may use an AST. These systems typically start with a near-source AST and systematically lower the level of abstraction until it is at or below the abstraction level of the target machine's ISA. The structure of that final, low-level AST tends to reflect the flow of values between operations.

AST-based systems usually use treewalks to traverse the IR. Many of the algorithms used in compilation have natural formulations as either a treewalk (see Section 11.4) or a depth-first search (see Section 8.5.1).

Some compilers build ASTs as derivative IRs because conversion into and out of an AST is fast and because it may simplify other algorithms. In particular, optimizations that rearrange expressions benefit from building an AST as a derivative IR because the AST eliminates all of the explicit names for intermediate results. Other algorithms, such as tree-height balancing (Section 8.4.2) or tree-pattern matching (Section 11.4) have "natural" expressions as tree traversals.

CHOOSING THE RIGHT ABSTRACTION

Even with a source level tree, representation choices affect usability. For example, the $\mathcal{R}^n$ Programming Environment used the subtree shown in panel (a) below to represent a complex number in FORTRAN, which was written as (c_1,c_2). This choice worked well for the syntax-directed editor, in which the programmer was able to change c_1 and c_2 independently; the pair node corresponded to the parentheses and the comma.



(a) AST Designed for Editing

(b) AST Designed for Compiling

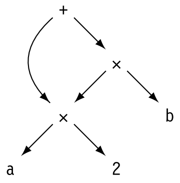
The pair format, however, proved problematic for the compiler. Each part of the compiler that dealt with constants needed special-case code for complex constants.

All other constants were represented with a single node that contained a pointer to the constant's lexeme, as shown above in panel (b). Using a similar format for complex constants would have complicated the editor, but simplified the compiler. Taken over the entire system, the benefits would likely have outweighed the complications.

Directed Acyclic Graphs

Directed acyclic graph
A DAG is an AST that represents each unique subtree once. DAGs are often called ASTs with sharing.

While an AST is more concise than a parse tree, it faithfully retains the structure of the original source code. For example, the AST for a × 2 + a × 2 × b contains two distinct copies of the expression a × 2. A *directed acyclic graph* (DAG) is a contraction of the AST that avoids this duplication. In a DAG, nodes can have multiple parents, and identical subtrees are reused. Such sharing makes the DAG more compact than the corresponding AST.



DAG for a × 2 + a × 2 × b

For expressions without assignment or function calls, textually identical expressions must produce identical values. The DAG for a × 2 + a × 2 × b, shown in the margin, reflects this fact by sharing a single copy of a × 2. If the value of a cannot change between the two uses of a, then the compiler should generate code to evaluate a × 2 once and use the result twice. This strategy can reduce the cost of evaluation. The DAG explicitly encodes the redundancy among subexpressions. If the compiler represents such facts in the IR, it can avoid the costs of rediscovering them.

When building the DAG for this expression, the compiler must prove that a's value cannot change between uses. If the expression contains neither

STORAGE EFFICIENCY AND GRAPHICAL REPRESENTATIONS

Many practical systems have used abstract syntax trees as their definitive IR. Many of these systems found that the AST was large relative to the size of the input text. In the $\mathcal{R}^n$ Programming Environment built at Rice in the 1980s, AST size posed two problems: mid-1980s workstations had limited memory, and tree–I/O slowed down all of the $\mathcal{R}^n$ tools.

$\mathcal{R}^n$ AST nodes used 92 bytes. The IR averaged 11 nodes per source-language statement. Thus, the AST needed about 1,000 bytes per statement. On a 4MB workstation, this imposed a practical limit of about 1,000 lines of code in most of the environment's tools.

No single decision led to this problem. To simplify memory allocation, $\mathcal{R}^n$ ASTs had only one kind of node. Thus, each node included all the fields needed by any node. (Roughly half the nodes were leaves, which need no pointers to children.) As the system grew, so did the nodes. New tools needed new fields.

Careful attention can combat this kind of node bloat. In $\mathcal{R}^n$, we built programs to analyze the contents of the AST and how it was used. We combined some fields and eliminated others. (In some cases, it was cheaper to recompute information than to write it and read it.) We used hash-linking to move rarely used fields out of the AST and into an ancillary table. (One bit in the node-type field indicated the presence of ancillary facts related to the node.) For disk I/O, we converted the AST to a linear form in a preorder treewalk, which made pointers implicit.

In $\mathcal{R}^n$, these changes reduced the size of ASTs in memory by about 75 percent. On disk, the files were about half the size of their memory representation. These changes let $\mathcal{R}^n$ handle larger programs and made the tools noticeably more responsive.

assignments nor calls to other procedures, the proof is easy. Since an assignment or a procedure call can change the value associated with a name, the DAG construction algorithm must invalidate subtrees when the values of their operands can change.

Mode of Use: DAGs are used in real systems for two primary reasons. If memory constraints limit the size of programs that the compiler can process, using a DAG as the definitive IR can reduce the IR's memory footprint. Other systems use DAGs to expose redundancies. Here, the benefit lies in better compiled code. These latter systems tend to use the DAG as a derivative IR—build the DAG, transform the definitive IR to reflect the redundancies, and discard the DAG.

4.3.2 **Graphs**

While trees provide a natural representation for the grammatical structure that parsing discovers in the source code, their rigid structure makes them less useful for representing other properties of programs. To model these aspects of program behavior, compilers often use more general graphs as IRs. The DAG introduced in the previous section is one example of a graph.

Control-Flow Graph

Basic block
a maximal length sequence of branch-free code

The simplest unit of control flow is a *basic block*—a maximal length sequence of straight-line, or branch-free, code. The operations in a block always execute together, unless some operation raises an exception. A block begins with a labeled operation and ends with a branch, jump, or predicated operation. Control enters a basic block at its first operation. The operations execute in an order consistent with top-to-bottom order in the block. Control exits at the block's last operation.

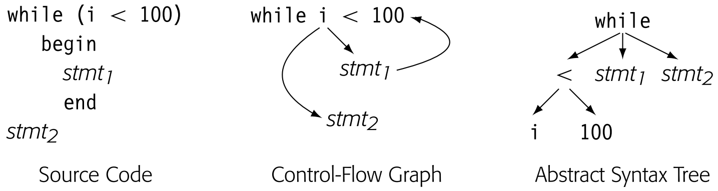
Control-flow graph
A CFG has a node for each basic block and an edge for each possible transfer of control.

A *control-flow graph* (CFG) models the flow of control between the basic blocks in a procedure. A CFG is a directed graph, $G = (N,E)$. Each node $n \in N$ corresponds to a basic block. Each edge $e = (n_i,n_j) \in E$ corresponds to a possible transfer of control from block n_i to block n_j.

If the compiler adds artificial entry and exit nodes, they may not correspond to actual basic blocks.

To simplify the discussion of program analysis in Chapters 8 and 9, we assume that each CFG has a unique entry node, n_0, and a unique exit node, n_f. If a procedure has multiple entries, the compiler can create a unique n_0 and add edges from n_0 to each actual entry point. Similarly, n_f corresponds to the procedure's exit. Multiple exits are more common than multiple entries, but the compiler can easily add a unique n_f and connect each of the actual exits to it.

The CFG provides a graphical representation of the possible runtime control-flow paths. It differs from syntax-oriented IRs, such as an AST, which show grammatical structure. Consider the while loop shown below. Its CFG is shown in the center pane and its AST in the rightmost pane.



```
while (i < 100)
   begin
     stmt1
     end
stmt2
```

Source Code

Control-Flow Graph

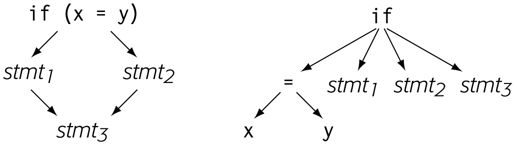
Abstract Syntax Tree

The CFG captures the essence of the loop: it is a control-flow construct. The cyclic edge runs from $stmt_1$ back to the test at the head of the loop. By contrast, the AST captures the syntax; it is acyclic but puts all the pieces in place to regenerate the source-code for the loop.

For an `if–then–else` construct both the CFG and the AST would be acyclic, as shown below.

```
if (x = y)
  then stmt1
  else stmt2
stmt3
```

Source Code



Control-Flow Graph

Abstract Syntax Tree

Again, the CFG models the control flow; one of $stmt_1$ or $stmt_2$ executes, but not both of them. The AST again captures the syntax but provides little direct intuition about how the code actually executes. Any such connection is implicit, rather than explicit.

Mode of Use: Compilers typically use a CFG in conjunction with another IR, making the CFG a derivative IR. The CFG represents the relationships among blocks, while the operations inside a block are represented with another IR, such as an expression-level AST, a DAG, or one of the linear IRs. A compiler could treat such a hybrid IR as its definitive IR, but the complications of keeping the multiple forms consistent makes this practice unusual.

Many parts of the compiler rely on a CFG, either explicitly or implicitly. Program analysis for optimization generally begins with control-flow analysis and CFG construction (see Chapter 9). Instruction schedulers need a CFG to understand how the scheduled code for individual blocks flows together (see Chapter 12). Register allocation relies on a CFG to understand how often each operation might execute and where to insert loads and stores for spilled values (see Chapter 13).

Section 4.4.4 covers CFG construction.

Block Length

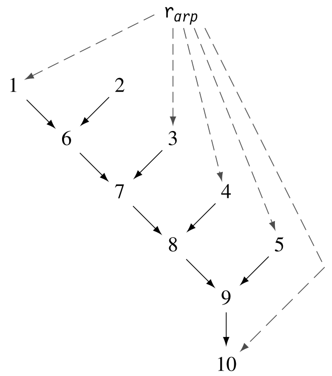
Some authors recommend building CFGs around blocks that are shorter than a basic block. The most common alternative block is a *single-statement block*. Single-statement blocks can simplify algorithms for analysis and optimization.

Single-statement blocks
a scheme where each block corresponds to a single source-level statement

The tradeoff between a CFG built with single-statement blocks and one built with maximal-length blocks involves both space and time. A CFG built on single-statement blocks has more nodes and edges than one built on maximal-length blocks. Thus, the single-statement CFG will use more memory than the maximal-length CFG, other factors being equal. With more

```
 1   loadAI   r_arp, @a  ⇒  r_a
 2   loadI    2          ⇒  r_2
 3   loadAI   r_arp, @b  ⇒  r_b
 4   loadAI   r_arp, @c  ⇒  r_c
 5   loadAI   r_arp, @d  ⇒  r_d
 6   mult     r_a, r_2   ⇒  r_a
 7   mult     r_a, r_b   ⇒  r_a
 8   mult     r_a, r_c   ⇒  r_a
 9   mult     r_a, r_d   ⇒  r_a
10   storeAI  r_a        ⇒  r_arp, @a
```

(a) Example Code from Chapter 1



(b) Dependence Graph for the Example

■ **FIGURE 4.3** An ILOC Basic Block and Its Dependence Graph.

nodes and edges, traversals take longer. More important, as the compiler annotates the CFG, the single-statement CFG has many more annotations than does the basic-block CFG. The time and space spent to build and use these annotations undoubtedly dwarfs the cost of CFG construction.

On the other hand, some optimizations benefit from single-statement blocks. For example, lazy code motion (see Section 10.3.1) only inserts code at block boundaries. Thus, single-statement blocks let lazy code motion optimize code placement at a finer granularity than would maximal-length blocks.

Dependence Graph

Compilers also use graphs to encode the flow of values from the point where a value is created, a *definition*, to any point where it is read, a *use*. A *data-dependence graph* embodies this relationship. Nodes in a data-dependence graph represent operations. Most operations contain both definitions and uses. An edge in a data-dependence graph connects two nodes, a definition in one and a use in the other. We draw dependence graphs with edges that run from the definition to the use; some authors draw these edges from the use to the definition.

Data-dependence graph
a graph that models the flow of values from definitions to uses in a code fragment

Definition
An operation that creates a value is said to *define* that value.

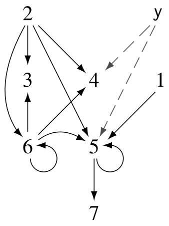
Use
An operation that references a value is called a *use* of that value.

Fig. 4.3 shows ILOC code to compute $a \leftarrow a \times 2 \times b \times c \times d$, also shown in Fig. 1.4. Panel (a) contains the ILOC code. Panel (b) shows the corresponding data-dependence graph.

The dependence graph has a node for each operation in the block. Each edge shows the flow of a single value. For example, the edge from 3 to 7 reflects the definition of r_b in statement 3 and its subsequent use in statement 7. The

```
1  x ← 0
2  i ← 1
3  while (i < 100)
4    if (y[i] > 0)
5       then x ← x + y[i]
6    i ← i + 1
7  print x
```

(a) The Code



(b) Its Dependence Graph

■ **FIGURE 4.4** Interaction Control Between Flow and the Dependence Graph.

virtual register r_{arp} contains an address that is at a fixed distance from the start of the local data area. Uses of r_{arp} refer to its implicit definition at the start of the procedure; they are shown with dashed lines.

The edges in the graph represent real constraints on the sequencing of operations—a value cannot be used until it has been defined. The dependence graph edges impose a partial order on execution. For example, the graph requires that 1 and 2 precede 6. Nothing, however, requires that 1 or 2 precedes 3. Many execution sequences preserve the dependences shown in the graph, including $\langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$ and $\langle 2, 1, 6, 3, 7, 4, 8, 5, 9, 10 \rangle$. The instruction scheduler exploits the freedom in this partial order, as does an "out-of-order" processor.

At a more abstract level, consider the code fragment shown in Fig. 4.4(a), which incorporates multiple basic blocks, along with both a `while` loop and an `if-then` construct. The compiler can construct a single dependence graph for the entire code fragment, as shown in panel (b).

References to `y[i]` derive their values from a single node that represents all of the prior definitions of `y`. Without sophisticated analysis of the subscript expressions, the compiler cannot differentiate between references to individual array elements.

This dependence graph is more complex than the previous example. Nodes 5 and 6 both depend on themselves; they use values that they may have defined in a previous iteration. Node 6, for example, can take the value of `i` from either 2 (in the initial iteration) or from itself (in any subsequent iteration). Nodes 4 and 5 also have two distinct sources for the value of `i`: nodes 2 and 6.

Mode of Use: Data-dependence graphs are typically built for a specific task and then discarded, making them a derivative IR. They play a central role in instruction scheduling (Chapter 12). They find application in a variety of optimizations, particularly transformations that reorder loops to expose parallelism and to improve memory behavior. In more sophisticated applications of the data-dependence graph, the compiler may perform extensive analysis of array subscript values to determine when references to the same array can overlap.

Call Graph

Interprocedural
Any technique that examines interactions across more than one procedure is called *interprocedural*.

Intraprocedural
Any technique that limits its attention to a single procedure is called *intraprocedural*.

Call graph
a graph that represents the calling relationships among the procedures in a program

The call graph has a node for each procedure and an edge for each call site.

To optimize code across procedure boundaries, some compilers perform *interprocedural* analysis and optimization. To represent calls between procedures, compilers build a *call graph*. A call graph has a node for each procedure and an edge for each distinct procedure call site. Thus, if the code calls q from three textually distinct sites in p, the call graph has three edges (p, q), one for each call site.

Both software-engineering practice and language features complicate the construction of a call graph.

Class hierarchy analysis
a static analysis that builds a model of a program's inheritance hierarchy

- Separate compilation limits the compiler's ability to build a call graph because it limits the set of procedures that the compiler can see. Some compilers build partial call graphs for the procedures in a compilation unit and optimize that subset.
- Procedure-valued parameters, both as actual parameters and as return values, create ambiguous calls that complicate call-graph construction. The compiler may perform an interprocedural analysis to limit the set of edges that such a call induces in the call graph, making call graph construction a process of iterative refinement. (This problem is analogous to the issue of ambiguous branches in CFG construction, as discussed in Section 4.4.4.)
- In object-oriented programs, inheritance can create ambiguous procedure calls that can only be resolved with additional type information. In some languages, class hierarchy analysis can disambiguate many of these calls; in others, that information cannot be known until runtime. Runtime resolution of ambiguous calls poses a serious problem for call graph construction; it also adds significant runtime overhead to the ambiguous calls.

Section 9.4 discusses some of the problems in call graph construction.

Mode of Use: Call graphs almost always appear as a derivative IR, built to support interprocedural analysis and optimization and then discarded. In fact, the best known interprocedural transformation, inline substitution (see Section 8.7.1), changes the call graph as it proceeds, rendering the old call graph inaccurate.

SECTION REVIEW

Graphical IRs present an abstract view of the code being compiled. The level of abstraction in a graphical IR, such as an AST, can vary from source level to below machine level. Graphical IRs can serve as definitive IRs or be built as special-purpose derivative IRs.

Because they are graphs, these IRs encode relationships that may be difficult to represent or manipulate in a linear IR. Graph traversals are an efficient way to move between logically connected points in the program; most linear IRs lack this kind of cross-operation connectivity.

REVIEW QUESTIONS

1. Given an input program, compare the expected size of the IR as a function of the number of tokens returned by the scanner for (a) a parse tree, (b) an AST, and (c) a DAG. Assume that the nodes in all three IR forms are of a uniform and fixed size.
2. How does the number of edges in a dependence graph for a basic block grow as a function of the number of operations in the block?

4.4 LINEAR IRS

Linear IRs represent the program as an ordered series of operations. They are an alternative to the graphical IRs described in the previous section. An assembly-language program is a form of linear code. It consists of an ordered sequence of instructions. An instruction may contain more than one operation; if so, those operations execute in parallel. The linear IRs used in compilers often resemble the assembly code for an abstract machine.

The logic behind using a linear form is simple. The source code that serves as input to the compiler is a linear form, as is the target-machine code that it emits. Several early compilers used linear IRs; this was a natural notation for their authors, since they had previously programmed in assembly code.

Linear IRs impose a total order on the sequence of operations. In Fig. 4.3, contrast the ILOC code with the data-dependence graph. The ILOC code has an implicit total order; the dependence graph imposes a partial order that allows multiple execution orders.

If a linear IR is used as the definitive IR in a compiler, it must include a mechanism to encode transfers of control among points in the program. Control flow in a linear IR usually models the implementation of control flow on the target machine. Thus, linear codes usually include both jumps and conditional branches. Control flow demarcates the basic blocks in a linear IR; blocks end at branches, at jumps, or just before labeled operations.

Taken branch
In most ISAs, conditional branches use only one label. Control flows either to the label, called the *taken branch*, or to the operation that follows the label, called the *fall-through branch*.

The fall-through path is often faster than the taken path.

Branches in ILOC differ from those found in a typical assembly language. They explicitly specify a label for both the taken path and the fall-through path. This feature eliminates all fall-through transfers of control and makes it easier to find basic blocks, to reorder basic blocks, and to build a CFG.

Many kinds of linear IRs have been used in compilers.

Destructive operation
an operation in which one of the operands is always redefined with the result

These operations likely arose as a way to save space in the instruction format on 8- or 16-bit machines.

- One-address codes model the behavior of accumulator machines and stack machines. These codes expose the machine's use of implicit names so that the compiler can tailor the code for it. The resulting IR can be quite compact.
- Two-address codes model a machine that has destructive operations. These codes fell into disuse as destructive operations and memory constraints on IR size became less important; a three-address code can model destructive operations explicitly.
- Three-address codes model a machine where most operations take two operands and produce a result. The rise of RISC architectures in the 1980s and 1990s made these codes popular again.

The rest of this section describes two linear IRs that are in common use: stack-machine code and three-address code. Stack-machine code offers a compact, storage-efficient representation. In applications where IR size matters, such as a JAVA applet transmitted over a network before execution, stack-machine code makes sense. Three-address code models the instruction format of a modern RISC machine; it has distinct names for two operands and a result. You are already familiar with one three-address code: the ILOC used throughout this book.

4.4.1 Stack-Machine Code

Stack-machine code, a form of one-address code, assumes the presence of a stack of operands. It is easy to generate and to understand. Most operations read their operands from the stack and push their results onto the stack. For example, a subtract operator removes the top two stack elements and pushes their difference onto the stack.

The stack discipline creates a need for new operations: `push` copies a value from memory onto the stack, `pop` removes the top element of the stack and writes it to memory, and `swap` exchanges the top two stack elements. Stack-based processors have been built; the IR seems to have appeared as a model for those ISAs. Stack-machine code for the expression `a - 2 × b` appears in the margin.

```
push  b
push  2
multiply
push  a
subtract
```

Stack Machine Code for `a - 2 × b`

Stack-machine code is compact. The stack creates an implicit name space and eliminates many names from the IR, which shrinks the IR. Using the stack, however, means that all results and arguments are transitory, unless the code explicitly moves them to memory.

Mode of Use: Stack-machine code is typically used as a definitive IR—often as the IR to transmit code between systems and environments. Both SMALLTALK-80 and JAVA use *bytecode*, the ISA of a stack-based virtual machine, as the external, interpretable form for code. The bytecode either runs in an interpreter, such as the JAVA virtual machine, or is translated into native target-machine code before execution. This design creates a system with a compact form of the program for distribution and a simple scheme for porting the language to a new machine: implement an interpreter for the virtual machine.

Bytecode
an IR designed specifically for its compact form; typically code for an abstract stack machine

The name derives from its limited size; many operations, such as multiply, need only a single byte.

4.4.2 Three-Address Code

In three-address code, most operations have the form `i ← j op k`, with an operator (`op`), two operands (`j` and `k`), and one result (`i`). Some operators, such as an immediate load and a jump, use fewer arguments. Sometimes, an operation has more than three addresses, such as a floating-point multiply-add operation. Three-address code for `a - 2 × b` appears in the margin. ILOC is a three-address code.

$$\begin{array}{l} t_1 \leftarrow 2 \\ t_2 \leftarrow b \\ t_3 \leftarrow t_1 \times t_2 \\ t_4 \leftarrow a \\ t_5 \leftarrow t_4 - t_3 \end{array}$$

Three-Address Code for `a - 2 × b`

Three-address code is attractive for several reasons. First, it is reasonably compact. Most operations consist of four items: an operation code, or *opcode*, and three names. The opcode and the names are drawn from limited sets. Opcodes typically require one or two bytes. Names are typically represented by integers or table indices. Second, separate names for the operands and the result give the compiler freedom to specify and control the reuse of names and values; three-address code has no destructive operations. Three-address code introduces a new set of compiler-generated names—names that hold the results of the various operations. A carefully chosen name space can reveal new opportunities to improve the code. Finally, since many modern processors implement three-address operations, a three-address code models their properties well.

Level of Abstraction: Within three-address codes, the set of supported operators and the level of abstraction can vary widely. Often, a three-address IR will contain mostly low-level operations, such as jumps, branches, loads, and stores, alongside more complex operations that encapsulate control flow, such as max. Representing these complex operations directly simplifies analysis and optimization.

For example, consider an operation that copies a string of characters from one address, the source, to another, the destination. This operation appeared as the bcopy library routine in the 4.2 BSD UNIX distribution and as the mvcl instruction (move character long) in the IBM 370 ISA. On a machine that does not implement an operation similar to mvcl, it may take many operations to perform such a copy.

IBM's PL.8 compiler, a pioneering RISC compiler, used this strategy.

Adding mvcl to the three-address code lets the compiler compactly represent this complex operation. The compiler can analyze, optimize, and move the operation without concern for its internal workings. If the hardware supports an mvcl-like operation, then code generation will map the IR construct directly to the hardware operation. If the hardware does not, then the compiler can translate mvcl into a sequence of lower-level IR operations or a call to a bcopy-like routine before final optimization and code generation.

Mode of Use: Compilers that use three-address codes typically deploy them as a definitive IR. Three-address code, with its explicit name space and its load-store memory model, is particularly well suited to optimization for register-to-register, load-store machines.

4.4.3 **Representing Linear Codes**

Many data structures have been used to implement linear IRs. The choices that a compiler writer makes affect the costs of various operations on IR code. Since a compiler spends most of its time manipulating the IR form of the code, these costs deserve some attention. While this discussion focuses on three-address codes, most of the points apply equally to stack-machine code (or any other linear form).

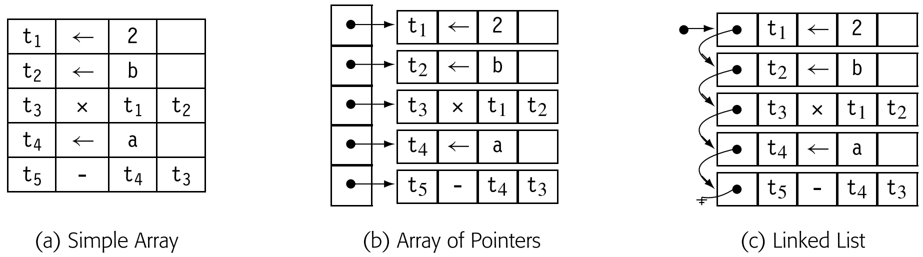
Three-address codes are often implemented as a set of quadruples. Each quadruple is represented with four fields: an operator, two operands (or sources), and a destination. To form blocks, the compiler needs a mechanism to connect individual quadruples. Compilers implement quadruples in a variety of ways.

$t_1 \leftarrow 2$
$t_2 \leftarrow b$
$t_3 \leftarrow t_1 \times t_2$
$t_4 \leftarrow a$
$t_5 \leftarrow t_4 - t_3$

Three-Address Code for $a - 2 \times b$

Fig. 4.5 shows three schemes to represent three-address code for $a - 2 \times b$ (shown in the margin). The first scheme, in panel (a), uses an array of structures. The compiler might build such an array for each CFG node to hold the code for the corresponding block. In panel (b), a vector of pointers holds the block's quadruples. Panel (c) links the quadruples together into a list.

Consider the costs incurred to rearrange the code in this block. The first operation loads a constant into a register; on most machines this translates



(a) Simple Array

(b) Array of Pointers

(c) Linked List

■ **FIGURE 4.5** Implementations of Three-Address Code for `a - 2 × b`.

directly into a load immediate operation. The second and fourth operations load values from memory, which on most machines might incur a multicycle delay unless the values are already in the primary cache. To hide some of the delay, the instruction scheduler might move the loads of `b` and `a` in front of the load immediate of 2.

In the array of structures, moving the load of `b` ahead of the immediate load requires saving the first operation to a temporary location, shuffling the second operation upward, and moving the immediate load into the second slot. The vector of pointers requires the same three-step approach, except that only the pointer values must be changed. The compiler can save the pointer to the immediate load, copy the pointer to the load of `b` into the first vector element, and rewrite the second vector element with the saved pointer. For the linked list, the operations are similar, except that the compiler must save enough state to let it traverse the list.

Now, consider what happens in the front end when it generates the initial round of IR. With the array of structures and the vector of pointers, the compiler must select a size for the array—in effect, the number of quadruples that it expects in a block. As it generates the quadruples, it fills in the data structure. If the compiler allocated too much space, that space is wasted. If it allocated too little, the compiler must allocate space for a larger array or vector, copy the contents into this new place, and free the original space. The linked list avoids these problems. Expanding the list just requires allocating a new quadruple and setting the appropriate pointer in the list.

A multipass compiler may use different implementations to represent the IR at different points in the compilation process. In the front end, where the focus is on generating the IR, a linked list might both simplify the implementation and reduce the overall cost. In an instruction scheduler, with its focus on rearranging the operations, the array of pointers might make more sense. A common interface can hide the underlying implementation differences.

INTERMEDIATE REPRESENTATIONS IN ACTUAL USE

In practice, compilers use a variety of IRs. Legendary FORTRAN compilers of yore, such as IBM's FORTRAN H compilers, used a combination of quadruples and control-flow graphs to represent the code for optimization. Since FORTRAN H was written in FORTRAN, it held the IR in an array.

For years, GCC relied on a very low-level IR, called register transfer language (RTL). GCC has since moved to a series of IRs. The parsers initially produce a language-specific, near-source tree. The compiler then lowers that tree to a second IR, GIMPLE, which includes a language-independent tree-like structure for control-flow constructs and three-address code for expressions and assignments. Much of GCC's optimizer uses GIMPLE; for example, GCC builds static single-assignment form (SSA) on top of GIMPLE. Ultimately, GCC translates GIMPLE into RTL for final optimization and code generation.

The LLVM compiler uses a single low-level IR; in fact, the name LLVM stands for "low-level virtual machine." LLVM's IR is a linear three-address code. The IR is fully typed and has explicit support for array and structure addresses. It provides support for vector or SIMD data and operations. Scalar values are maintained in SSA form until the code reaches the compiler's back end. In LLVM environments that use GCC front ends, LLVM IR is produced by a pass that performs GIMPLE-to-LLVM translation.

The Open64 compiler, an open-source compiler for the IA-64 architecture, used a family of five related IRs, called WHIRL. The parser produced a near-source-level WHIRL. Subsequent phases of the compiler introduced more detail to the WHIRL code, lowering the level of abstraction toward the actual machine code. This scheme let the compiler tailor the level of abstraction to the various optimizations that it applied to IR.

Notice that some information is missing from Fig. 4.5. For example, no labels are shown because labels are a property of the block rather than any individual quadruple. Storing a list of labels with the CFG node for the block saves space in each quadruple; it also makes explicit the property that labels occur only at the start of a block. With labels attached to a block, the compiler can ignore them when reordering operations inside the block, avoiding one more complication.

4.4.4 Building the CFG from Linear Code

Compilers often must convert between different IRs, often different styles of IRs. One routine conversion is to build a CFG from a linear IR such as ILOC. The essential features of a CFG are that it identifies the beginning and end of each basic block and connects the resulting blocks with edges that

```
FindLeaders()
    next ← 1
    Leader[next++] ← 1
    create a CFG node for $l_1$
    for i ← 2 to n do
        if $op_i$ has a label $l_i$ then
            Leader[next++] ← i
            create a CFG node for $l_i$
    // MaxStmt is a global variable
    MaxStmt ← next - 1
```

(a) Finding Leaders

```
BuildGraph()
    for i ← 1 to MaxStmt do
        j ← Leader[i] + 1
        while (j ≤ n and $op_j$ ∉ Leader) do
            j ← j + 1
        j ← j - 1
        Last[i] ← j
        if $op_j$ is "cbr $r_k$ → $l_1$, $l_2$" then
            add edge from j to node for $l_1$
            add edge from j to node for $l_2$
        else if $op_j$ is "jumpI → $l_1$" then
            add edge from j to node for $l_1$
        else if $op_j$ is "jump → $r_1$" then
            add edges from j to all labeled
                statements
        end
```

(b) Finding Last and Adding Edges

■ **FIGURE 4.6** Building a Control-Flow Graph.

describe the possible transfers of control among blocks. Often, the compiler must build a CFG from a simple, linear IR that represents a procedure.

As a first step, the compiler must find the start and the end of each basic block in the linear IR. We will call the initial operation of a block a *leader*. An operation is a leader if it is the first operation in the procedure, or if it has a label that is, potentially, the target of some branch. The compiler can identify leaders in a single pass over the IR, shown in Fig. 4.6(a). *FindLeaders* iterates over the operations in the code, in order, finds the labeled statements, and records them as leaders.

If the linear IR contains labels that are not used as branch targets, then treating labels as leaders may unnecessarily split blocks. The algorithm could track which labels are jump targets. However, ambiguous jumps may force it to treat all labeled statements as leaders.

Ambiguous jump
a branch or jump whose target is not known at compile time (e.g., a jump to an address in a register)

In ILOC, `jump` is ambiguous while `jumpI` is not.

The second pass, shown in panel (b), finds every block-ending operation. It assumes the ILOC model where every block, except the final block, ends with a branch or a jump and branches specify labels for both the taken and not-taken paths. This assumption simplifies the handling of blocks and allows the compiler's optimizer or back end to choose which path will be the "fall through" case of a branch. (For the moment, assume branches have no delay slots.)

CFG construction with fall-through branches is left as an exercise for the reader (see Exercise 4).

To find the end of each block, the algorithm iterates through the blocks, in order of their appearance in the *Leader* array. It walks forward through the IR until it finds the leader of the next block. The operation immediately before that leader ends the current block. The algorithm records that operation's index in *Last[i]*, so that the pair ⟨*Leader[i], Last[i]*⟩ describes block i. It adds edges to the CFG as needed.

For a variety of reasons, the CFG should have a unique entry node n_0 and a unique exit node n_f. If the underlying code does not have this shape, a simple postpass over the graph can create n_0 and n_f.

Complications in CFG Construction

Features of the IR, the target machine, and the source language can complicate CFG construction.

Pseudooperation
an operation that manipulates the internal state of the assembler or compiler, but does not translate into an executable operation

Ambiguous jumps may force the compiler to add edges that are not feasible at runtime. The compiler writer can improve this situation by recording the potential targets of ambiguous jumps in the IR. ILOC includes the `tbl` pseudooperation to specify possible targets of an ambiguous jump (see Appendix A). Anytime the compiler generates a `jump`, it should follow the `jump` with one or more `tbl` operations that record the possible targets. The hints reduce spurious edges during CFG construction.

PC-relative branch
A transfer of control that specifies an offset, either positive or negative, from its own memory address.

If a tool builds a CFG from target-machine code, features of the target ISA can complicate the process. The algorithm in Fig. 4.6 assumes that all leaders, except the first, are labeled. If the target machine has fall-through branches, the algorithm must be extended to recognize unlabeled statements that receive control on a fall-through path. PC-relative branches cause a similar set of complications.

Branch delay slots introduce complications. The compiler must group any operation in a delay slot into the block that preceded the branch or jump. If that operation has a label, it is a member of multiple blocks. To disambiguate such an operation, the compiler can place an unlabeled copy of the operation in the delay slot and use the labeled operation to start the new block.

If a branch or jump can occur in a branch delay slot, the CFG builder must walk forward from the leader to find the block-ending branch—the first branch it encounters. Branches in the delay slot of a block-ending branch can be pending on entry to the target block. In effect, they can split the target block into multiple blocks and create both new blocks and new edges. This feature adds serious complications to CFG construction.

Some languages allow jumps to labels outside the current procedure. In the procedure that contains the jump, the jump target can be modeled with a new block. In the procedure that contains the target, however, the labeled block can pose a problem. The compiler must know that the label is the target of a nonlocal jump; otherwise, analysis passes may produce misleading results. For this reason, languages such as PASCAL or ALGOL restricted nonlocal jumps to visible labels in outer lexical scopes. C requires the use of the functions `setjmp` and `longjmp` to expose these transfers.

SECTION REVIEW

Linear IRs represent the code being compiled as an ordered sequence of operations. Linear IRs vary in their level of abstraction; the source code for a program in a text file is a linear form, as is the assembly code for that same program. Linear IRs lend themselves to compact, human-readable representations.

Two widely used linear IRs are bytecodes, generally implemented as a one-address code with implicit names on many operations, and three-address code, similar to ILOC.

REVIEW QUESTIONS

1. Consider the expression $a \times 2 + a \times 2 \times b$. Translate it into stack-machine code and into three address code. Compare and contrast the total number of operations and operands in each form. How do they compare to the tree in Fig. 4.2(b)?
2. Sketch the modifications that must be made to the algorithm in Fig. 4.6 to account for ambiguous jumps and branches. If all jumps and branches are labeled with a construct similar to ILOC's `tbl`, does that simplify your algorithm?

4.5 SYMBOL TABLES

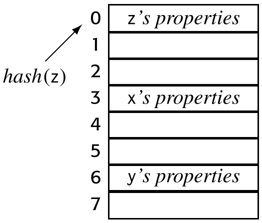
During parsing the compiler discovers the names and properties of many distinct entities, including named values, such as variables, arrays, records, structures, strings, and objects; class definitions; constants; labels in the code; and compiler-generated temporary values (see the digression on page 192).

Symbol table
a collection of one or more data structures that hold information about names and values

Most compilers maintain symbol tables as persistent ancillary data structures used in conjunction with the IR that represents the executable code.

For each name actually used in the program, the compiler needs a variety of information before it can generate code to manipulate that entity. The specific information will vary with the kind of entity. For a simple scalar variable the compiler might need a data type, size, and storage location. For a function it might need the type and size of each parameter, the type and size of the return value, and the relocatable assembly-language label of the function's entry point.

Thus, the compiler typically creates one or more repositories to store descriptive information, along with an efficient way to locate the information associated with a given name. Efficiency is critical because the compiler will consult these repositories many times.



The compiler typically keeps a set of tables, often referred to as *symbol tables*. Conceptually, a symbol table has two principal components: a map from textual name to an index in a repository and a repository where that index leads to the name's properties. An abstract view of such a table is shown in the margin.

A compiler may use multiple tables to represent different kinds of information about different kinds of values. For names, it will need a symbol table that maps each name to its properties, declared or implicit. For aggregates, such as records, arrays, and objects, the compiler will need a structure table that records the entity's layout: its constituent members or fields, their properties, and their relative location within the structure. For literal constants, such as numbers, characters, and strings, the compiler will need to lay out a data area to hold these values, often called a constant pool. It will need a map from each literal constant to both its type and its offset in the pool.

Constant pool
a statically initialized data area set aside for constant values

4.5.1 Name Resolution

The primary purpose of a symbol table is to resolve names. If the compiler finds a reference to name *n* at some point *p* in a program, it needs a mechanism that maps *n* back to its declaration in the naming environment that holds at *p*. The map from name to declaration and properties must be well defined; otherwise, a program might have multiple meanings. Thus, programming languages introduce rules that specify where a given declaration of a name is both valid and visible.

Scope
the region of a program where a given name can be accessed

In general, a scope is a contiguous set of statements in which a name is declared, visible, and accessible. The limits of a scope are marked by specific symbols in the language. Typically, a new procedure defines a new scope that covers its entire definition. C and C++ demarcate blocks with curly braces. Each block defines a new scope.

In most languages, scopes can nest. A declaration for *x* in an inner scope obscures any definitions of *x* in surrounding scopes. Nesting creates a hierarchy of name spaces. These hierarchies play a critical role in software

REPRESENTING REFERENCES IN THE IR

In the implementation of an IR, the compiler writer must decide how to represent a reference to a source language name. The compiler could simply record the lexeme; that decision, however, will require a symbol-table lookup each time that the compiler uses the reference.

The best alternative may be to store a handle to the relevant symbol table reference. That handle could be an absolute pointer to the table entry; it might be a pointer to the table and an offset within the table. Such a handle will allow direct access to the symbol table information; it should also enable inexpensive equality tests.

engineering; they allow the programmer to choose local names without concern for their use elsewhere in the program.

The two most common name-space hierarchies are created by lexical scope rules and inheritance rules. The compiler must build tables to model each of these hierarchies.

Lexical Scopes

A lexical-scoping environment uses properly nested regions of code as scopes. A name n declared in scope s is visible inside s. It is visible inside any scope nested in s, with the caveat that a new declaration of n obscures any declaration of n from an outer scope.

At a point p in the code, a reference to n maps to the first declaration of n found by traversing the scopes from the scope containing the reference all the way out to the *global scope*. Lexically scoped languages differ greatly in the depth of nesting that they allow and the set of scopes that they provide. (Sections 5.4.1, 6.3.1, and 6.4.3 discuss lexical scopes in greater depth.)

Global scope
an outer scope for names visible in the entire program

Inheritance Hierarchies

Object-oriented languages (OOLs) introduce another set of scopes: the inheritance hierarchy. OOLs create a data-centric naming scheme for objects; objects have data and code members that are accessed relative to the object rather than relative to the current procedure.

In an OOL, explicitly declared subclass and superclass relationships define the inheritance hierarchy—a naming regime similar to the lexical hierarchy and orthogonal to it. Conceptually, subclasses nest within superclasses, just as inner scopes nest within outer scopes in a lexical hierarchy. The compiler builds tables to model subclass and superclass relationships, as well.

Superclass and Subclass
In a language with inheritance, if class x inherits members and properties from class y, we say that x is a *subclass* of y and y is the *superclass* of x.

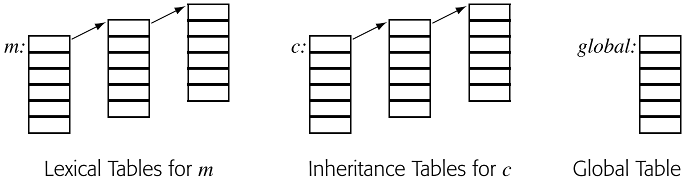
The terminology used to specify inheritance varies across languages. In JAVA, a subclass *extends* its superclass. In C++, a subclass is *derived* from its superclass.

Hierarchical Tables

The compiler can link together tables, built for individual scopes, to represent the naming hierarchies in any specific input program. A typical program in an Algol-like language (ALL) might have a single linked set of tables to represent the lexically nested scopes. In an OOL, that lexical hierarchy would be accompanied by another linked set of tables to represent the inheritance hierarchy.

When the compiler encounters a reference in the code, it first decides whether the name refers to a variable (either global or local to some method) or an object member. That determination is usually a matter of syntax; languages differentiate between variable references and object references. For a variable reference, it begins in the table for the current scope and searches upward until it finds the reference. For an object member, it determines the object's class and begins a search through the inheritance hierarchy.

In a method *m*, declared in some class *c*, the search environment might look as follows. We refer to this scheme as the "sheaf of tables."



The lookup begins in the table for the appropriate scope and works its way up the chain of tables until it either finds the name or exhausts the chain. Chaining tables together in this fashion creates a flexible and powerful tool to model the complex scoping environments of a modern programming language.

The compiler writer can model complex scope hierarchies—both lexical hierarchies and inheritance hierarchies—with multiple tables linked together in a way that reflects the language-designated search order. For example, nested classes in JAVA give rise to a lexical hierarchy within the inheritance hierarchy. The link order may vary between languages, but the underlying technology need not change.

In a modern environment, the compiler will likely retain each of these tables for later use, either in the compiler or in related tools such as performance monitors or debuggers. The sheaf-of-tables approach can create compact, separable tables suitable for preservation.

Other Scopes

Other language features create nested scopes. For example, records, structures, and objects all act as independent scopes. Each creates a new name space and has a set of visibility rules. The compiler may or may not choose to implement these scopes with a full-blown sheaf in a hash table; nonetheless each is a distinct scope and can be modeled with a new sheaf in the appropriate table. The constant pool might also be treated as a separate scope.

4.5.2 Table Implementation

As we have seen, a compiler will contain multiple distinct tables, ranging from symbol tables and inheritance tables through structure layout tables and constant tables. For each table, the compiler writer must choose an appropriate implementation strategy: both a mapping function and a repository. While the choices are, for the most part, independent, the compiler writer may want to use the same basic strategies across multiple tables so that they can share implementations.

Implementing the Mapping

The mapping from a textual name to an index can be implemented in myriad ways, each with their own advantages and disadvantages.

Linear List

A linear list is simple to construct, to expand, and to search. The primary disadvantage of a linear list is that searching the list takes $\mathsf{O}(n)$ time per lookup, where n is the number of items in the list. Still, for a small procedure, a linear list might make sense.

Tree

A tree structure has the advantages of a list, including simple and efficient expansion. It also has the potential to significantly reduce the time required per lookup. Assuming that the tree is roughly balanced—that is, the subtrees at each level are approximately the same size—then the expected case lookup time should approach $\mathsf{O}(\log_2 n)$ per item, where n is the number of items in the tree.

Balanced trees use more complex insertion and deletion protocols to maintain roughly equal-sized subtrees. The literature contains many effective and efficient techniques for building balanced trees.

Unbalanced trees have simpler insertion and deletion protocols but provide no guarantees on the relative sizes of sibling subtrees. An unbalanced tree can devolve into linear search when presented with an adversarial input.

Hash Map

Hash collision
When two strings map to the same table index, we say that they *collide*. For hash function $h(x)$ and table size s, if

$$h(x) \bmod s = h(y) \bmod s,$$

then x and y will collide.

The compiler can use a numerical computation, called a hash, to produce an integer from a string. A well-designed hash function, h, distributes those integers so that few strings produce the same hash value. To build a hash table, the programmer uses the hash value of a string, modulo the table size, as an index into a table.

Handling collisions is a key issue in hash table design, as discussed in Appendix B.4. If the set of keys produces no collisions, then insertion and lookup in a hash table should take $O(1)$ time. If the set of keys all map to the same table index, then insertion and lookup might devolve to $O(|keys|)$ time per lookup. To avoid this problem, the compiler writer should use a well-designed hash function, as found in a good algorithms textbook.

Static Map

As an alternative to hashing, the compiler can precompute a collision-free static map from keys to indices. Multiset discrimination solves this problem (see the digression on page 190).

For small sets of keys, an approach that treats the keys as a set of acyclic regular expressions and incrementally builds a DFA to recognize that set can lead to a fast implementation (see Section 2.6.2). Once the transition-table size exceeds the size of the level-one data cache, this approach slows down considerably.

Implementing the Repository

The implementation of the repository storage for the information associated with a given name can be orthogonal to the lookup mechanism. Different tables may need distinct and different structures to accommodate the kinds of information that the compiler needs. Nonetheless, these repositories should have some common properties (see also Appendix B.4).

Block contiguous
These allocators use two protocols: a major allocation obtains space for multiple records while a minor one returns a single record. Minor allocations use a fast method; they amortize the major allocation cost over many records.

- Record storage should be either contiguous or block-contiguous to improve locality, decrease allocation costs, and simplify reading and writing the tables to external media.
- Each repository should contain enough information to rebuild the lookup structure, in order to accommodate graceful table expansion, and to facilitate restoring the structures from external media.

- The repository should support changes to the search path. For example, as the parser moves in and out of different scopes, the search path should change to reflect the current situation.

From a management perspective, the repository must be expandable in order to handle large programs efficiently without wasting space on small ones. Its index scheme should be independent of the mapping scheme so that the map can be expanded independently of the repository; ideally, the map will be sparse and the repository dense.

SECTION REVIEW

Compilers build ancillary data structures to augment the information stored in the compiler's definitive IR. The most visible of these structures is a set of symbol tables that map a name in the source text or the IR into the set of properties associated with that name.

This section explored several issues in the design of these ancillary tables. It showed how linking tables together in explicit search paths can model both lexical scope rules and inheritance rules. It discussed tradeoffs in the implementation of both the mapping mechanism and the repository for these tables.

REVIEW QUESTIONS

1. Using the "sheaf-of-tables" scheme, what is the complexity of inserting a new name into the table at the current scope? What is the complexity of looking up a name in the set of tables? How deep are the lexical and inheritance hierarchies in programs that you write?
2. When the compiler initializes a scope, it likely needs an initial symbol table size. How might the parser estimate that initial symbol table size? How might later passes of the compiler estimate it?

4.6 NAME SPACES

Most discussions of name spaces focus on the source program's name space: lexical scopes and inheritance rules. Of equal importance, from the perspective of the quality of compiled code, is the name space created in the compiler's IR. A specific naming discipline can either expose opportunities for optimization or obscure them. The choices that the compiler makes with regard to names determine, to a large extent, which computations can be analyzed and optimized.

AN ALTERNATIVE TO HASHING

Symbol tables are often implemented with hash maps, due to the expected efficiency of hashing. If the compiler writer is worried about the unlikely but possible worst-case behavior of hashing, multiset discrimination provides an interesting alternative. It avoids the possibility of worst-case behavior by constructing the index offline, in the scanner.

To use multiset discrimination, the compiler first scans the entire program and builds a ⟨*name,pos*⟩ tuple for each instance of an identifier, where *name* is the identifier's lexeme and *pos* is its ordinal position in the list of classified words, or *tokens*. It enters all the tuples into a large set.

Next, the compiler sorts the set lexicographically. In effect, this creates a set of subsets, one per identifier. Each subset holds the tuples for all the occurrences of its identifier. Since each tuple refers to a specific token, through its *position* value, the compiler can use the sorted set to modify the token stream. The compiler makes a linear scan over the set, processing each subset. It allocates a symbol-table index for each unique identifier, then rewrites the tokens to include that index. The parser can read symbol-table indices directly from the tokens. If the compiler needs a textual lookup function, the resulting table is ordered alphabetically for a binary search.

This technique adds some cost to compilation. It makes an extra pass over the token stream, along with a lexicographic sort. In return, it avoids any possibility of worst-case behavior from hashing and it makes the initial size of the symbol table obvious before parsing begins. This technique can replace a hash table in almost any application in which an offline solution will work.

The IR name space is intimately related to the memory model used in translation. The compiler may assume that all values are kept in memory, except when they are actively used in a computation. The compiler may assume that values are kept in registers whenever possible. If the compiler uses stack-machine code as its IR, it will keep these active values on the stack. These different assumptions radically affect the set of values that can be named, analyzed, and optimized.

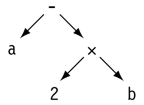
This section focuses on issues in name space design; it also introduces one important example: *static single assignment form*. The next section explores the issues that arise in choosing a memory model.

4.6.1 Name Spaces in the IR

When compiler writers design an IR, they should also design a naming discipline for the compiler to follow. The choice of a name space interacts with the choice of an IR; some IRs allow broad latitude in naming, while others make most names implicit in the representation.

Implicit Versus Explicit Names

Tree-like IRs use implicit names for some values. Consider an AST for a - 2 × b, shown in the margin. It has nodes for each of a, 2, b, 2 × b, and a - 2 × b. The interior nodes, those for 2 × b and a - 2 × b, lack explicit names that the compiler can manipulate.



AST for a - 2 × b

By contrast, three-address code uses only explicit names, which gives the compiler control over the naming discipline. It can assign explicit names to any or all of the values computed in the code. Consider, for example, the ILOC code for a - 2 × b, shown in the margin. The upper version introduces a unique name for every unknown value and expression—register names r_0 through r_3. After execution, each of those values survives in its own register. The lower version uses a different naming discipline intended to conserve names. After it executes, the two quantities that survive are a in r_0 and a - 2 × b in r_1.

```
load   @b       ⇒ r0
multI  2, r0    ⇒ r1
load   @a       ⇒ r2
subI   r2, r1   ⇒ r3
```

ILOC for a - 2 × b
With Unique Names

```
load   @b       ⇒ r0
multI  2, r0    ⇒ r1
load   @a       ⇒ r0
subI   r0, r1   ⇒ r1
```

ILOC for a - 2 × b
With Name Reuse

The example makes it appear that graphical IRs use implicit names and linear IRs use explicit names. It is not that simple. Stack-machine code relies on an implicit stack data structure, which leads to implicit names for many values. A CFG has explicit names for each of the nodes so that they can be connected to the corresponding code fragments. Even an AST can be rewritten to include explicit names; for any expression or subexpression that the compiler wants to name, it can insert an assignment and subsequent use for a compiler-generated name.

Variables Versus Values

In the source program, the set of accessible names is determined by the source language rules and the details of the program. Declared named variables are visible; intermediate subexpressions are not. In the statement:

```
a ← 2 * b + cos(c / 3)
```

a, b, and c can each be used in subsequent statements. The values of 2 * b, c / 3, and cos(c / 3) cannot.

In the IR form of the code, the compiler writer must decide which values to expose via consistent, explicit names. The compiler can use as many names as necessary; compiler writers sometimes refer to these generated names as *virtual names*. The compiler might translate the statement so that the code evaluates each of these three expressions into its own unique name. Alternatively, by reusing names, it could eliminate any chance for reuse.

Virtual name
A compiler-generated name is often called a *virtual name*, in the same sense as *virtual memory* or a *virtual register*.

THE IMPACT OF NAMING

In the late 1980s, we experimented with naming schemes in a FORTRAN 77 compiler. The first version generated a new name for each computation; it simply bumped a counter to get a new name. This approach produced large name spaces; for example, 985 names for a 210-line implementation of the singular value decomposition (SVD). Objectively, this name space seemed large. It caused speed and space problems in the register allocator, where name space size determines the size of many data structures. (Today, we have better data structures, and much faster machines with more memory.)

The second version used an allocate/free protocol to manage names. The front end allocated temporary names on demand and freed them when the immediate uses were finished. This scheme shrank the name space; SVD used roughly 60 names. Allocation was faster; for example, the time to compute LIVEOUT sets for SVD decreased by 60 percent (see Section 8.6.1).

Unfortunately, reuse of names obscured the flow of values and degraded the quality of optimization. The decline in code quality overshadowed any compile-time benefits.

Further experimentation led to a short set of rules that yielded strong optimization while mitigating growth in the name space.

1. Each textual expression received a unique name, found by hashing. Thus, each occurrence of an expression, for example, $r_{17} + r_{21}$, targeted the same register.
2. In $\langle \texttt{op} \rangle\ r_i, r_j \Rightarrow r_k$, k was chosen so that $i, j < k$.
3. Register copy operations, $r_i \Rightarrow r_j$, were allowed to have $i > j$ only if r_j corresponded to a declared scalar variable. Registers for variables were only defined by copy operations. Expressions were evaluated into their "natural" register and then were moved into the register for the variable.
4. Each store operation, $r_i \Rightarrow \texttt{MEM}(r_j)$, was followed by a copy from r_i into the variable's named register. (Rule 1 ensures that loads from that location always target the same register. Rule 4 ensures that the virtual register and memory location contain the same value.)

With this name space, the compiler used about 90 names for SVD. It exposed all of the optimizations found with the first name-space scheme. The compiler used these rules until we adopted the SSA name space.

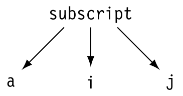
This decision has a direct effect on what the compiler can do in subsequent optimization.

The compiler writer enforces these decisions by codifying them in the translation rules that generate the IR. These decisions have a widespread effect on the efficiency of both the compiler and the code that it generates.

The temptation, of course, is to provide a unique name for each subexpression, so as to maximize the opportunities presented to the optimizer. However, not all subexpressions are of interest. A value that is only live in a single block does not need a name that persists through the entire procedure. Exposing such a value to procedure-wide analysis and optimization is unlikely to change the code.

The converse is true, as well. Any value that is live in multiple blocks or is computed in multiple blocks may merit an explicit, persistent name. Expressions that are computed in multiple blocks, on multiple paths, are prime targets for a number of classical global optimizations. Providing a single consistent name across multiple definitions and uses can expose an expression to analysis and transformations that improve the compiled code (see Chapters 8–10).

Finally, the choice of a naming discipline also interacts with decisions about the level of abstraction in the IR. Consider again the two representations of an array reference, a[i,j], shown in the margin. The source-level AST, along with the symbol table, contains all of the essential information needed to analyze or translate the reference. (The symbol table will contain a's type, data area, and offset in that data area along with the number of dimensions and their upper and lower bounds.) The corresponding ILOC code exposes more details of the address calculations and provides explicit names for each subexpression in that calculation.



Source-Level Tree

```
subI   r_i, 1    ⇒ r_1
multI  r_1, 10   ⇒ r_2
subI   r_j, 1    ⇒ r_3
add    r_2, r_3  ⇒ r_4
multI  r_4, 4    ⇒ r_5
loadI  @a        ⇒ r_6
add    r_5, r_6  ⇒ r_7
load   r_7       ⇒ r_aij
```

ILOC Code

These two representations expose and name different quantities. The AST explicitly exposes the fact that the calculation computes the address for a[i,j], but shows no details of that calculation. The ILOC code exposes the fact that the address computation involves seven distinct subexpressions, any one of which might occur elsewhere. The question of which IR is better depends entirely on how the compiler writer intends to use the information.

4.6.2 Static Single-Assignment Form

Static single-assignment form (SSA) is an IR and a naming discipline that many modern compilers use to encode information about both the flow of control and the flow of values in the program. In SSA form, each name corresponds to one definition point in the code. The term *static single assignment* refers to this fact. As a corollary, each use of a name in an operation encodes information about where the value originated; the textual name refers to a specific definition point.

SSA form
an IR that has a value-based name system, created by renaming and use of pseudooperations called ϕ-functions

SSA encodes both control and value flow. It is used widely in optimization (see Section 9.3).

A program is in SSA form when it meets two constraints: (1) each definition has a distinct name; and (2) each use refers to a single definition. To transform an IR program to SSA form, the compiler inserts *ϕ-functions* at points

ϕ-function
A ϕ-function takes several names and merges them, defining a new name.

```
x ← ···
y ← ···
while (x < 100)
  x ← x + 1
  y ← y + x
```

(a) Original Code

```
      x0 ← ···
      y0 ← ···
      if (x0 ≥ 100) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < 100) goto loop
next: x3 ← φ(x0,x2)
      y3 ← φ(y0,y2)
```

(b) Code in SSA Form

■ **FIGURE 4.7** A Small Loop in SSA Form.

where different control-flow paths merge and it then renames variables so that the single-assignment property holds.

To clarify the impact of these rules, consider the small loop shown in Fig. 4.7(a). Panel (b) shows the same code in SSA form. Variable names include subscripts to create a distinct name for each definition. ϕ-functions have been inserted at points where multiple distinct values can reach the start of a block. Finally, the `while` construct has been rewritten at a lower level of abstraction to expose the fact that the initial test refers to x_0 while the end-of-loop test refers to x_2.

The code shape for the loop is discussed in Section 7.5.2.

The ϕ-function has an unusual semantics. It acts as a copy operation that selects, as its argument, the value that corresponds to the edge along which control entered the block. Thus, when control flows into the loop from the block above the loop, the ϕ-functions at the top of the loop body copy the values of x_0 and y_0 into x_1 and y_1, respectively. When control flows into the loop from the test at the loop's bottom, the ϕ-functions select their other arguments, x_2 and y_2.

The execution semantics of ϕ-functions are different than other operations. On entry to a block, all its ϕ-functions read the value of their appropriate argument, in parallel. Next, they all define their target SSA names, in parallel. Defining their behavior in this way allows the algorithms that manipulate SSA form to ignore the ordering of ϕ-functions at the top of a block—an important simplification. It does, however, complicate the process of translating SSA form back into executable code, as discussed in Section 9.3.5.

The definition of SSA form prevents two ϕ-functions from defining the same SSA name.

SSA form was intended for analysis and optimization. The placement of ϕ-functions in SSA form encodes information about the creation and use of individual values. The single-assignment property of the name space allows the compiler to ignore many issues related to the lifetimes of values;

Lifetime
For a value a, its lifetime spans from its first definition to its last use.

BUILDING SSA

Static single-assignment form is the only IR we describe that does not have an obvious construction algorithm. Section 9.3 presents one construction algorithm in detail. However, a sketch of the construction process will clarify some of the issues. Assume that the input program is already in ILOC form. To convert it to an equivalent linear form of SSA, the compiler must first insert ϕ-functions and then rename the ILOC virtual registers.

The simplest way to insert ϕ-functions is to find each block that has multiple CFG predecessors and add a ϕ-function for each ILOC virtual register at the start of that block. This process inserts many unneeded ϕ-functions; most of the complexity in the full algorithm focuses on eliminating those extraneous ϕ-functions.

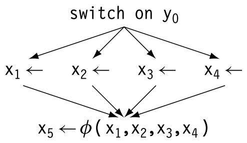
To rename the ILOC virtual registers, the compiler can process the blocks, in a depth-first order. For each virtual register, it keeps a counter. When the compiler encounters a definition of r_i, it increments the counter for r_i, say to k, and rewrites the definition with the name r_{i_k}. As the compiler traverses the block, it rewrites each use of r_i with r_{i_k} until it encounters another definition of r_i. (That definition bumps the counter to k + 1.) At the end of a block, the compiler looks down each control-flow edge and rewrites the appropriate ϕ-function parameter for r_i in each block that has multiple predecessors.

After renaming, the code conforms to the two rules of SSA form. Each definition creates a unique name. Each use refers to a single definition. Several better SSA construction algorithms exist; they insert fewer ϕ-functions than this simple approach.

for example, because names are not redefined, the value of a name is available along any path that proceeds from that operation. These two properties simplify and improve many optimizations.

The example exposes some oddities of SSA form that bear explanation. Consider the ϕ-function that defines x_1. Its first argument, x_0, is defined in the block that precedes the loop. Its second argument, x_2, is defined later in the block labeled loop. Thus, when the ϕ first executes, one of its arguments is undefined. In many programming-language contexts, this would cause problems. Since the ϕ-function reads only one argument, and that argument corresponds to the most recently taken edge in the CFG, it can never read the undefined value.

A ϕ-function takes an arbitrary number of operands. To fit SSA form into a three-address IR, the compiler writer must include mechanisms to accommodate longer operand lists and to associate those operands with specific incoming edges. Consider the block at the end of a case statement as shown in the margin.



ϕ-Function at the End of a Case Statement

The ϕ-function for x_5 must have an argument for each case. The number of arguments it needs is bounded only by the number of paths that enter the block. Thus, an operation to represent that ϕ-function in a linear IR will need an arbitrary number of arguments. It does not fit directly into the fixed-arity, three-address scheme.

In a simple array representation for three-address code, the compiler writer will need a side data structure to hold ϕ-function arguments. In the other two schemes for implementing three-address code shown in Fig. 4.5, the compiler can insert tuples of varying size. For example, the tuples for load and load immediate might have space for just two names, while the tuple for a ϕ-operation could be large enough to accommodate all its operands, plus an operand count.

SECTION REVIEW

The compiler must generate internal names for all the values computed in a program. Those names may be explicit or implicit. The rules used to generate names directly affect the compiler's ability to analyze and optimize the IR. Careful use of names can encode and expose facts for later use in optimization. Proliferation of names enlarges data structures and slows compilation.

The SSA name space encodes properties that can aid in analysis and optimization; for example, it lets optimizations avoid the need to reason about redefinitions of names (see Section 8.4.1). This additional precision in naming can both simplify algorithms and improve the optimizer's results.

REVIEW QUESTIONS

1. The ILOC code shown in the margin on page 193 uses as many virtual register names as practical—assuming that r_i and r_j cannot be renamed because they represent variables in the program. Construct an equivalent code that uses as few virtual names as possible.
2. Convert the code shown in the margin to SSA form, following the explanation on page 195. Does each ϕ-functions that you inserted serve a purpose?

```
read x, y, z;

while (x > 0) {
    y = y * z;
    x = x - 1;
}

print y;
```

4.7 PLACEMENT OF VALUES IN MEMORY

Almost every IR has an underlying storage map. The compiler must assign a storage location to each value that the compiled code computes or uses. That location might be in a register or in memory. It might be a specific location: a physical register or ⟨*base address*,*offset*⟩ pair. It might be a symbolic location: a virtual register or a symbolic label. The location's lifetime must match the lifetime of the value; that is, it must be available and dedicated to the value from the time the value is created until the time of its last use.

This section begins with a discussion of memory models—the implicit rules used to assign values to data areas. The later subsections provide more detail on data area assignment and layout.

4.7.1 Memory Models

Before the compiler can translate the source program into its IR form, the compiler must understand, for each value computed in the code, where that value will reside. The compiler need not enumerate all the values and their locations, but it must have a mechanism to make those decisions consistently and incrementally as translation proceeds. Typically, compiler writers make a set of decisions and then apply them throughout translation. Together, these rules form a memory model for the compiled code.

Memory models help define the underlying model of computation: where does an operation find its arguments? They play a critical role in determining which problems the compiler must solve and how much space and time it will take to solve them.

Three memory models are common: a *memory-to-memory* model, a *register-to-register* model, and a *stack* model. These models share many characteristics; they differ in where they store values that are active in the current computation.

Active value
A value is *active* in the immediate neighborhood where it is used or defined.

Memory-to-Memory Model Values have their primary home in memory. Either the IR supports memory-to-memory operations, or the code moves active values into registers and inactive values back to memory.

Register-to-Register Model Whenever possible, values are kept in a *virtual register*; some local, scalar values have their only home in a virtual register. Global values have their homes in memory (see Section 4.7.2).

Stack Model Values have their primary home in memory. The compiler moves active values onto and off of the stack with explicit operations (e.g., `push` and `pop`). Stack-based IRs and ISAs often include operations to reorder the stack (e.g., `swap`).

```
add @a, @b ⇒ @c
```

```
load  @a         ⇒ vr_i
load  @b         ⇒ vr_j
add   vr_i, vr_j ⇒ vr_k
store vr_k       ⇒ @c
```

(a) Memory-to-Memory Model

```
add vr_a, vr_b ⇒ vr_c
```

(b) Register-to-Register Model

```
push @b
push @a
add
pop  ⇒ @c
```

(c) Stack Model

■ **FIGURE 4.8** Three-Operand Add Under Different Memory Models.

Unambiguous value
A value that can be accessed with just one name is *unambiguous*.

Ambiguous value
Any value that can be accessed by multiple names is *ambiguous*.

Fig. 4.8 shows the same `add` operation under each of these models. Panel (a) shows the operation under two different assumptions. The left column assumes that the `add` takes memory operands, shown as symbolic labels. The right column assumes that the `add` is a register-to-register operation, with values resident in memory. The choice between these two designs probably depends on the target machine's ISA. Panel (b) shows the same `add` in a register-to-register model. It assumes that `a`, `b`, and `c` are all unambiguous scalar values that reside in virtual registers: vr_a, vr_b, and vr_c, respectively. Panel (c) shows the operation under a stack model; it assumes that the variable's home locations are in memory and named by symbolic labels.

These distinct memory models have a strong impact on the shape of the IR code and on the priorities for the optimizer and back end.

- In a memory-to-memory model, the unoptimized form of the code may use just a few registers. That situation places a premium on optimizations that promote values into unused registers for nontrivial parts of their lifetimes. In the back end, register allocation focuses more on mapping names than on reducing demand for physical registers.
- In a register-to-register model, the unoptimized code may use many more virtual registers than the target machine supplies. That situation encourages optimizations that do not significantly increase demand for registers. In the back end, register allocation is required for correctness and is one of the key determiners of runtime performance.
- In a stack model, the structure of the target machine becomes critical. If the ISA has stack operations, as does the JAVA virtual machine, then optimization focuses on improving the stack computation. If the ISA is a CISC or RISC processor, then the compiler will likely translate the stack-machine code into some other form for code generation.

The JAVA HotSpot server compiler translated JAVA bytecode to a graphical IR for optimization and code generation.

In the end, the choice of memory model has a strong influence on the design of the compiler's optimizer and back end.

THE HIERARCHY OF MEMORY OPERATIONS IN ILOC 9X

Under any memory model, the compiler writer should look for ways to encode more facts about values into the IR. In the 1990s, we built a research compiler that used an IR named ILOC 9x. The IR featured a hierarchy of memory operations that allowed the compiler to encode knowledge about values kept in memory. At the bottom of the hierarchy, the compiler had little or no knowledge about the value; at the top of the hierarchy, it knew the actual value. These operations are as follows:

Operation	Meaning
Immediate load	Loads a known constant value into a register.
Nonvarying load	Loads a value that does not change at runtime. The compiler does not know the value but can prove that the program does not change it.
Scalar load & store	Operate on a scalar value, not an array element, a structure element, or a pointer-based value.
Generic load & store	Operate on a value that may vary and may be non-scalar. It is the general-case operation.

With this hierarchy, the front end encoded knowledge about the target value directly into the ILOC 9x code. Other passes could rewrite operations from a more general to a more restricted form as they discovered new facts. For example, if the compiler discovered that a load always produced a known constant value, it replaced the generic or scalar load with an immediate load.

Optimizations capitalized on the facts encoded in this way. For example, a comparison between the result of a nonvarying load and a constant must itself be invariant—a fact that might be difficult or impossible to prove with a generic load operation.

4.7.2 Keeping Values in Registers

With a register-to-register memory model, the compiler tries to assign as many values as possible to virtual registers. This approach relies heavily on the register allocator to map virtual registers in the IR to physical registers in the final code, and to *spill* to memory any virtual register that it cannot keep in a physical register.

Spill
A register allocator *spills* a value by storing it to a designated location in memory. It may later *restore* the value to a register.

The compiler cannot keep an ambiguous value in a register across an assignment. With an unambiguous value *x*, the compiler knows precisely where *x*'s value changes: at assignments to *x*. Thus, the compiler can safely generate code that keeps *x* in a register.

With an ambiguous value *x*, however, an assignment to some other ambiguous value *y* might change *x*'s value. If the compiler tries to hold *x* in a register across an assignment to *y*, the register may not be updated with the new value. To make matters worse, in a given procedure, *x* and *y* might refer to the same storage location in some invocations and not in others. This situation makes it difficult for the compiler to generate correct code that keeps *x* in a register. Relegating *x* to memory lets the addressing hardware resolve which assignments should change *x* and which should not.

If a call passes a global name to a call-by-reference parameter, the callee can access the value with either its global name or the formal parameter name.

The same effect occurs when a call passes a name *x* in two different call-by-reference parameter slots.

In practice, compilers decide which values they consider unambiguous, and relegate all ambiguous values to storage in memory—one of the data areas or the heap—rather than in a register. Ambiguity can arise in multiple ways. Values stored in pointer-based variables are often ambiguous. Call-by-reference parameters can be ambiguous. Many compilers treat array-element values as ambiguous because the compiler cannot tell if two references, such as `A[i,j]` and `A[m,n]` can ever refer to the same element.

Typically, compilers focus on proving that a given value is unambiguous. The analysis might be cursory and local. For example, in C, any local variable whose address is never taken is unambiguous. More complex analyses build sets of possible names for each pointer variable; any variable whose set has just one element is unambiguous. Analysis cannot resolve all ambiguities; the unprovable cases are treated as if they were proven to be ambiguous.

Language features can affect the compiler's ability to analyze ambiguity. For example, ANSI C includes two keywords that directly communicate information about ambiguity. The `restrict` keyword informs the compiler that a pointer is unambiguous. It is often used when a procedure passes an address directly at a call site. The `volatile` keyword lets the programmer declare that the contents of a variable may change arbitrarily and without notice. It is used for hardware device registers and for variables that might be modified by interrupt service routines or other threads of control in an application.

4.7.3 Assigning Values to Data Areas

Data area
A region in memory set aside to hold data values. Each data area is associated with some specific scope.

Examples include local data areas for procedures and global data areas.

Just as the compiler must choose a name for each value in the program, so, too, must it decide where those values will reside at runtime. While the memory model determines where values live while they are active, each of the memory models discussed in Section 4.7.1 consigns some values to memory when they are not active. The compiler must decide, for each such value, where it should reside during its lifetime.

Most temporary values will live in the space reserved for active values—either registers or memory locations in the local data area—as determined by both the memory model and the availability of space. For variables that are declared in the source program, the compiler assigns each one a permanent home, based on its individual properties: its lifetime, its visibility, and its declaring scope.

Lifetime A value's *lifetime* refers to the period of time during which its value can be defined or referenced. Outside of a value's lifetime, it is undefined.

Region of Visibility A value is *visible* if it can be named—that is, the code can read or write the value. Its region of visibility is, simply, the code in which it is visible.

Declaring Scope A variable's lifetime and visibility depend on the scope that declares it. For example, a file static variable in C has a lifetime of the entire execution; it is only visible inside the file that declares it.

Programming languages have rules that determine lifetime, visibility, and scope for each name.

To simplify memory management, most compilers create a set of *data areas* associated with various program scopes. For memory resident variables, the combination of lifetime, visibility, and declaring scope determines which data area will hold the variable's value.

From a storage layout perspective, the compiler will categorize lifetimes into one of three categories.

Automatic An automatic variable's lifetime matches one activation of its scope (a procedure or block). The value is defined and used inside the scope and its value ceases to exist on exit from the scope. A local variable is, typically, automatic by default.

Automatic
A name whose lifetime matches a single activation of the scope that declares it is an *automatic* variable.

We call these variables "automatic" because their allocation and deallocation can be handled as part of entry and exit for the corresponding scope. At runtime, each invocation of a procedure has its own local data area where automatic variables can be stored.

Static A static variable's lifetime might span multiple activations of its declaring scope. If it is assigned a value, that value persists after control exists the scope where the assignment occurred.

Static
A name that retains its value across multiple activations of its scope is a *static* variable.

The compiler can allocate such variables once, before execution; they are, in effect, always present. Static variables are stored in a preallocated data area associated with the declaring scope. The compiler may combine the static data areas for multiple scopes.

	Scope	Lifetime	Location
ALLs	Local	Automatic	Registers or local data area of declaring scope
	Local	Static	Procedure or file static data area
	File	Static	File static data area
	Global	Static	Global data area
OOLs	Method	Automatic	Registers or local data area of declaring scope
	Method	Static	Class record or method-specific static data area
	Class	Static	Class record
	Global	Static	Global data area
	any scope	Irregular	Explicitly allocated on the heap

■ **FIGURE 4.9** Variable Placement by Scope and Lifetime.

Constant values are a special case; they are static values that can be initialized with an assembly-level directive. The compiler typically creates a separate data area for them, often called a constant pool.

Irregular
An entity whose lifetime depends on explicit allocation and either explicit or implicit deallocation is an *irregular* entity.

Irregular An irregular variable has a lifetime that is not tied to any single scope. It is, typically, allocated explicitly; it may be freed either explicitly or implicitly. Examples include objects in JAVA and strings created with `malloc` in C.

Variables with irregular lifetimes are, in general, allocated space on the runtime heap (see Section 5.6.1).

The compiler can categorize each value by its lifetime and scope. This classification suggests a specific data area for the value's storage. Fig. 4.9 shows a typical scheme that a compiler might use to place variables into registers and data areas.

Given a mapping from values to data areas, the compiler must assign each memory-resident value a location. It iterates over the data areas and, within a data area, over the values for that data area. It assigns each value a specific offset from the start of the data area. Algorithms for this assignment are discussed in Section 5.6.3.

For values that might be kept in registers, the compiler assigns them a virtual register name. The actual assignment of virtual registers to hardware registers is left to the register allocator.

SECTION REVIEW

The compiler must determine, for each value that the program computes, where that value will reside at runtime. The compiler determines those locations based on the programming language, on the memory model adopted by the compiler, on lifetime information for the values, and on the compiler writer's knowledge of the target machine's system architecture. The compiler systematically assigns each value to a register or a data area and assigns offsets within data areas to individual values.

Decisions about the placement of values can affect the performance of compiled code. Storage layout can change the locality behavior of the program. Storage assignment decisions can encode subtle knowledge about properties of the underlying code, such as the ambiguity of values.

REVIEW QUESTIONS

1. Consider the function `fib` shown in the margin. Write down the ILOC that a compiler's front end might generate for this code using a register-to-register model and using a memory-to-memory model. How does the code for the two models compare?
2. Write the pseudocode for an algorithm that takes a list of variables assigned to some data area and assigns them offsets. Explain what information the compiler needs for each variable.

```
int fib(int n) {
  int x = 0;
  int y = 1;
  int z = 0;
  while (n > 0) {
     z = x + y;
     x = y;
     y = z;
     n = n - 1;
  }
  return z;
}
```

4.8 SUMMARY AND PERSPECTIVE

The choice of an IR has a major impact on the design, implementation, speed, and effectiveness of a compiler. None of the intermediate forms described in this chapter are, definitively, the right answer for all compilers or all tasks in a given compiler. The compiler writer must consider the overall goals of a compiler project when selecting an IR, designing its implementation, and adding ancillary data structures such as symbol and label tables.

Contemporary compiler systems use all manner of IRs, ranging from parse trees and abstract syntax trees (often used in source-to-source systems) through lower-than-machine-level linear codes (used, for example, in GCC). Many compilers use multiple IRs—building a second or third one to perform a particular analysis or transformation, then modifying the original, and definitive, one to reflect the result.

CHAPTER NOTES

The literature on IRs and experience with them is sparse. Nonetheless, IRs have a major impact on both the structure and behavior of a compiler. The classic IR forms, such as syntax trees, ASTs, DAGs, quadruples, triples, and one-address code have been described in textbooks since the 1970s [8, 36,157,181]. Newer IR forms like SSA [56,120,279] are described in the literature on analysis and optimization. The original JAVA HotSpot Server compiler used a form of program dependence graph as its definitive IR [92]. Muchnick discusses IRs in detail and highlights the use of multiple levels of IR in a single compiler [279].

The observation that multiple passes over the code can lead to more efficient code dates back to Floyd [160]; this fundamental observation creates the need for IR and justifies the added expense of the multipass compiler. This insight applies in many contexts within a compiler.

The idea of using a hash function to recognize textually identical operations dates back to Ershov [150]. Its specific application in Lisp systems seems to appear in the early 1970s [135,174]; by 1980, it was common enough that McCarthy mentions it without citation [267].

Cai and Paige introduced multiset discrimination as an alternative to hashing [71]. Their intent was to create an efficient lookup mechanism with guaranteed constant time behavior. Closure-free regular expressions, described in Section 2.6.2, can achieve a similar effect. The work on shrinking the size of $\mathcal{R}^n$'s AST was done by David Schwartz and Scott Warren.

EXERCISES

Section 4.3

1. Both a parse tree and an abstract syntax tree retain information about the form of the source program.
   a. What is the relationship between the size of the parse tree and the size of the input program?
   b. What is the relationship between the size of the abstract syntax tree and the size of the input program?
   c. What relationship would you expect between the size of the parse tree and the size of the abstract syntax tree? In other words, what value would you expect for $\frac{|parse\ tree|}{|abstract\ syntax\ tree|}$?
2. Write an algorithm to convert an expression tree into a DAG.

```
procedure main
     integer a, b, c;
     procedure f1(w,x);
         integer a,x,y;
         call f2(w,x);
         end;
     procedure f2(y,z)
         integer a,y,z;
         procedure f3(m,n)
             integer b, m, n;
Here ------>     c = a * b * m * n;
             end;
         call f3(c,z);
         end;
     ...
     call f1(a,b);
     end;
```

■ **FIGURE 4.10** Program for Exercise 5.

Section 4.4

3. Consider the following code fragment. Show how it might be represented in an abstract syntax tree, in a control-flow graph, and in three-address code.

```
if (c[i] ≠ 0)
    then a[i] ← b[i] ÷ c[i];
    else a[i] ← b[i];
```

   Discuss the advantages of each representation. For what applications would one representation be preferable to the others?

4. The algorithm for constructing a CFG, shown in Fig. 4.6, assumes that the conditional branch operation, `cbr`, specifies a label for both the taken branch and the fall-through branch.

   Modify both *FindLeaders* and *BuildGraph* to handle input code where the `cbr` operation only specifies the taken branch.

Section 4.5

5. You are writing a compiler for a simple lexically scoped language. Consider the example program shown in Fig. 4.10.

   a. Draw the symbol table and its contents just before the line of code indicated by the arrow.

   b. For each name mentioned in the statement indicated by the arrow, show which declaration defines it.

6. Consider the code fragment shown in Fig. 4.11. Draw its CFG.

```
x ← ···
y ← ···
a ← y + 2
b ← 0
while (x < a)
    if (y < x) then
        x ← y + 1
        y ← b × 2
    else
        x ← y + 2
        y ← a ÷ 2;
    w ← x + 2
    z ← y × a
    y ← y + 1
```

■ **FIGURE 4.11** Code Fragment for Exercise 6.

Section 4.6

7. Write both three-address code and stack-machine code to evaluate the expression $a \times (b + c) \div d$. Assume that the IR can represent a load of a's value with a load from the label @a.

   a. How many names does the three-address code use?

   b. How many names does the stack-machine code use?

8. Three-address code and two-address code differ in the ways that the operations interact with the name space. With three-address code, overwriting a value in some name *n* is a choice. With two-address code, ordinary arithmetic operations such as add overwrite one of the two arguments. Thus, with two-address code, the compiler must choose which operands to preserve and which operands to overwrite.

   Write down three ways that the compiler might encode the expression $a \leftarrow b \times c$ into a low-level two-address code. Assume that b and c reside in r_b and r_c before the multiply operation.

   How might the compiler choose between these different encodings of the operation into two-address code?

Section 4.7

9. Consider the three C procedures shown in Fig. 4.12.

   a. In a compiler that uses a register-to-register memory model, which variables in procedures A, B, and C would the compiler be forced to store in memory? Justify your answer.

   b. Suppose the compiler uses a memory-to-memory model. Consider the execution of the two statements that are in the if clause of the if–else construct. If the compiler has two registers available at that point in the computation, how many loads and stores would the

```
static int max = 0;

void A(int b, int e)
{
  int a, c, d, p;

  a = B(b);
  if (b > 100) {
     c = a + b;
     d = c * 5 + e;
  }
  else
     c = a * b;

  p = c;
  C(&p);
}
```

```
int B(int k)
{
  int x, y;

  x = pow(2, k);
  y = x * 5;
  return y;
}


void C(int *p)
{
  if (*p > max)
     max = *p;
}
```

■ **FIGURE 4.12** Code for Exercise 9.

compiler need to issue in order to load values in registers and store them back to memory during execution of those two statements? What if the compiler has three registers available?

10. In FORTRAN, two variables can be forced to begin at the same storage location with an `equivalence` statement. For example, the following statement forces `a` and `b` to share storage:

```
equivalence (a,b)
```

Can the compiler keep a local variable in a register throughout the procedure if that variable appears in an equivalence statement? Justify your answer.

This page intentionally left blank

Chapter 5

Syntax-Driven Translation

ABSTRACT

The compiler's task is to translate the input program into a form where it can execute directly on the target machine. The scanner and parser can analyze the code presented for compilation and determine whether that code constitutes a well-formed program in the source language. If the compiler is to perform translation, optimization, and code generation, however, it must build an IR version of the code for the compiler's later use. That process requires the compiler to reason about the code at a level that is deeper than the context-free syntax.

This chapter looks at some of the problems that arise in performing that first translation from source code to IR in the compiler's front end, along with the mechanisms used to solve those problems. It focuses on a particular style of syntax-driven computation that was popularized by parser generators—a style that has become common practice.

KEYWORDS

Syntax-Driven Translation, IR Generation, Symbol Tables

5.1 INTRODUCTION

Fundamentally, the compiler is a program that reads in another program, builds a representation of its meaning, analyzes and improves the code in that form, and translates the code so that it executes on some target machine. Translation, analysis, optimization, and code generation require an in-depth understanding of the input program. The purpose of *syntax-driven translation* is to begin to assemble the knowledge that later stages of compilation will need.

As a compiler parses the input program, it builds an IR version of the code. It annotates that IR with facts that it discovers, such as the type and size of a variable, and with facts that it derives, such as where it can store each value. Compilers use two mechanisms to build the IR and its ancillary data structures: (1) syntax-driven translation, a form of computation embedded into the parser and sequenced by the parser's actions, and (2) subsequent traversals of the IR to perform more complex computations.

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00011-5

Conceptual Roadmap

The primary purpose of a compiler is to translate code from the source language into the target language. This chapter explores the mechanism that compiler writers use to translate a source-code program into an IR program. The compiler writer plans a translation, at the granularity of productions in the source-language grammar, and tools execute the actions in that plan as the parser recognizes individual productions in the grammar. The specific sequence of actions taken at compile time depends on both the plan and the parse.

During translation, the compiler develops an understanding, at an operational level, of the source program's meaning. The compiler builds a model of the input program's name space. It uses that model to derive information about the type of each named entity. It also uses that model to decide where, at runtime, each value computed in the code will live. Taken together, these facts let the compiler emit the initial IR program that embodies the meaning of the original source code program.

A Few Words About Time

Translation exposes all of the temporal issues that arise in compiler construction. At design time, the compiler writer plans both runtime behavior and compile-time mechanisms to create code that will elicit that behavior. She encodes those plans into a set of syntax-driven rules associated with the productions in the grammar. Still at design time, she must reason about both compile-time support for translation, in the form of structures such as symbol tables and processes such as type checking, and runtime support to let the code find and access values. (We will see that support in Chapters 6 and 7, but the compiler writer must think about how to create, use, and maintain that support while designing and implementing the initial translation.)

Runtime system
the routines that implement abstractions such as the heap and I/O

At compiler-build time, the parser generator turns the grammar and the syntax-driven translation rules into an executable parser. At compile time, the parser maps out the behaviors and bindings that will take effect at runtime and encodes them in the translated program. At runtime, the compiled code interacts with the runtime system to create the behaviors that the compiler writer planned back at design time.

Overview

The compiler writer creates a tool—the compiler—that translates the input program into a form where it executes directly on the target machine. The compiler, then, needs an implementation plan, a model of the name space, and a mechanism to tie model manipulation and IR generation back to the structure and syntax of the input program. To accomplish these tasks:

- The compiler needs a mechanism that ties its information gathering and IR-building processes to the syntactic structure and the semantic details of the input program.
- The compiler needs to understand the visibility of each name in the code—that is, given a name *x*, it must know the entity to which *x* is bound. Given that binding, it needs complete type information for *x* and an access method for *x*.
- The compiler needs an implementation scheme for each programming language construct, from a variable reference to a case statement and from a procedure call to a heap allocation.

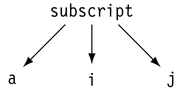
This chapter focuses on a mechanism that is commonly used to specify syntax-driven computation. The compiler writer specifies actions that should be taken when the parser reduces by a given production. The parser generator arranges for those actions to execute at the appropriate points in the parse. Compiler writers use this mechanism to drive basic information gathering, IR generation, and error checking at levels that are deeper than syntax (e.g., does a statement reference an undeclared identifier?).

Section 5.3 introduces a common mechanism used to translate source code into IR. It describes, as examples, implementation schemes for expression evaluation and some simple control-flow constructs. Section 5.4 explains how compilers manage and use symbol tables to model the naming environment and track the attributes of names. Section 5.5 introduces the subject of type analysis; a complete treatment of type systems is beyond the scope of this book. Finally, Section 5.6 explores how the compiler assigns storage locations to values.

Chapters 6 and 7 discuss implementation of other common programming language constructs.

5.2 BACKGROUND

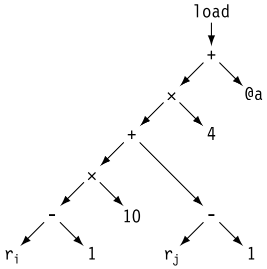
The compiler makes myriad decisions about the detailed implementation of the program. Because the decisions are cumulative, compiler writers often adopt a strategy of progressive translation. The compiler's front end builds an initial IR program and a set of annotations using information available in the parser. It then analyzes the IR to infer additional information and refines the details in the IR and annotations as it discovers and infers more information.



Near-Source AST

To see the need for progressive translation, consider a tree representation of an array reference `a[i,j]`. The parser can easily build a relatively abstract IR, such as the near-source AST shown in the margin. The AST only encodes facts that are implicit in the code's text.

To generate assembly code for the reference, the compiler will need much more detail than the near-source AST provides. The low-level tree shown in the margin exposes that detail and reveals a set of facts that cannot be seen in the near-source tree. All those facts play a role in the final code.



Low-Level Tree

- The compiler must know that `a` is a 10 × 10 array of four-byte integers with lower bounds of 1 in each dimension. Those facts are derived from the statements that declare or create `a`.
- The compiler must know that `a` is stored in row-major order (see Fig. 5.16). That fact was decided by the language designer or the compiler writer before the compiler was written.
- The compiler must know that `@a` is an assembly-language label that evaluates to the runtime address of the first element of `a` (see Section 7.3). That fact derives from a naming strategy adopted at design time by the compiler writer.

To generate executable code for `a[i,j]`, the compiler must derive or develop these facts as part of the translation process.

This chapter explores both the mechanism of syntax-driven translation and its use in the initial translation of code from the source language to IR. The mechanism that we describe was introduced in an early LR(1) parser generator, `yacc`. The notation allows the compiler writer to specify a small snippet of code, called an *action*, that will execute when the parser reduces by a specific production in the grammar.

Syntax-driven translation lets the compiler writer specify the action and relies on the parser to decide when to apply that action. The syntax of the input program determines the sequence in which the actions occur. The actions can contain arbitrary code, so the compiler writer can build and maintain complex data structures. With forethought and planning, the compiler writer can use this syntax-driven translation mechanism to implement complex behaviors.

Through syntax-driven translation, the compiler develops knowledge about the program that goes beyond the context-free syntax of the input code. Syntactically, a reference to a variable *x* is just a name. During translation, the compiler discovers and infers much more about *x* from the contexts in which the name appears.

- The source code may define and manipulate multiple distinct entities with the name *x*. The compiler must map each reference to *x* back to the appropriate runtime instance of *x*; it must bind *x* to a specific entity based on the naming environment in which the reference occurs. To do so, it builds and uses a detailed model of the input program's name space.
- Once the compiler knows the binding of *x* in the current scope, it must understand the kinds of values that *x* can hold, their size and structure, and their lifetimes. This information lets the compiler determine what operations can apply to *x*, and prevents improper manipulation of *x*. This knowledge requires that the compiler determine the type of *x* and how that type interacts with the contexts in which *x* appears.
- To generate code that manipulates *x*'s value, the compiler must know where that value will reside at runtime. If *x* has internal structure, as with an array, structure, string, or record, the compiler needs a formula to find and access individual elements inside *x*. The compiler must determine, for each value that the program will compute, where that value will reside.

To complicate matters, executable programs typically include code compiled at different times. The compiler writer must design mechanisms that allow the results of the separate compilations to interoperate correctly and seamlessly. That process begins with syntax-driven translation to build an IR representation of the code. It continues with further analysis and refinement. It relies on carefully planned interactions between procedures and name spaces (see Chapter 6).

5.3 SYNTAX-DRIVEN TRANSLATION

Syntax-driven translation is a collection of techniques that compiler writers use to tie compile-time actions to the grammatical structure of the input program. The front end discovers that structure as it parses the code. The compiler writer provides computations that the parser triggers at specific points in the parse. In an LR(1) parser, those actions are triggered when the parser performs a reduction.

5.3.1 A First Example

Fig. 5.1(a) shows a simple grammar that describes the set of positive integers. We can use syntax-driven actions tied to this grammar to compute the value of any valid positive integer.

Panel (b) contains the *Action* and *Goto* tables for the grammar. The parser has three possible reductions, one per production.

1	*Number*	→	*DList*
2	*DList*	→	*DList* `digit`
3		|	`digit`

(a) The Positive Integer Grammar

	Action		Goto
State	`eof`	`digit`	*DList*
0		s 2	1
1	acc	s 3	
2	r 3	r 3	
3	r 2	r 2	

(b) Its *Action* and *Goto* Tables

■ **FIGURE 5.1** The Grammar for Positive Integers.

- The parser reduces by rule 3, *DList* → `digit`, on the leftmost digit in the number.
- The parser reduces by rule 2, *DList* → *DList* `digit`, for each digit after the first digit.
- The parser reduces by rule 1, *Number* → *DList* after it has already reduced the last `digit`.

This strategy has been used by assembly language programmers for decades.

The parser can compute the integer's value with a series of production-specific tasks. It can accumulate the value left to right and, with each new digit, multiply the accumulated value by ten and add the next digit. Values are associated with each symbol used in the parse. We can encode this strategy into production-specific rules that are applied when the parser reduces.

Using the notation popularized by the parser generators `yacc` and `bison`, the rules might be:

```
Number  : DList         { return $1; } ;
DList   : DList digit   { $$ = $1 * 10 + CToI($2); } ;
        | digit         { $$ = CToI($1); } ;
```

The symbols `$$`, `$1`, and `$2` refer to values associated with grammar symbols in the production. `$$` refers to the nonterminal symbol on the rule's left-hand side (LHS). The symbol `$`*i* refers to the value for the *i*th symbol on the rule's right-hand side (RHS).

The example assumes that `CToI()` converts the character from the lexeme to an integer. The compiler writer must pay attention to the types of the stack cells represented by `$$`, `$1`, and so on.

Recall that the initial $ on the stack represents the pair ⟨*INVALID*, *INVALID*⟩.

Using the *Action* and *Goto* tables from Fig. 5.1(b) to parse the string "`175`", an LR(1) parser would take the sequence of actions shown in Fig. 5.2. The reductions, in order, are: *reduce 3*, *reduce 2*, *reduce 2*, and *accept*.

Iteration	State	Word	Stack	Action
0	–	1	$ ⟨*Number* 0⟩	*shift 2*
1	2	7	$ ⟨*Number* 0⟩ ⟨digit 2⟩	*reduce 3*
2	1	7	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩	*shift 3*
3	3	5	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩ ⟨digit 3⟩	*reduce 2*
4	1	5	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩	*shift 3*
5	3	eof	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩ ⟨digit 3⟩	*reduce 2*
6	1	eof	$ ⟨*Number* 0⟩ ⟨*DList* 1⟩	*accept*

■ **FIGURE 5.2** Parser Actions for the Number "175".

- *Reduce 3* applies rule 3's action with the integer 1 as the value of `digit`. The rule assigns one to the LHS *DList*.
- *Reduce 2* applies rule 2's action, with 1 as the RHS *DList*'s value and the integer 7 as the `digit`. It assigns $1 \times 10 + 7 = 17$ to the LHS *DList*.
- *Reduce 2* applies rule 2's action, with 17 as the RHS *DList*'s value and 5 as the `digit`. It assigns $17 \times 10 + 5 = 175$ to the LHS *DList*.
- The accept action, which is also a reduction by rule 1, returns the value of the LHS *DList*, which is 175.

The reduction rules, applied in the order of actions taken by the parser, create a simple framework that computes the integer's value.

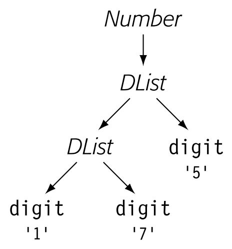
The critical observation is that the parser applies these rules in a predictable order, driven by the structure of the grammar and the parse of the input string. The compiler writer specifies an action for each reduction; the sequencing and application of those actions depend entirely on the grammar and the input string. This kind of syntax-driven computation forms a programming paradigm that plays a central role in translation and finds many other applications.

Of course, this example is overkill. A real system would almost certainly perform this same computation in a small, specialized piece of code, similar to the one in the margin. It implements the same computation, without the overhead of the more general scanning and parsing algorithms. In practice, this code would appear inline rather than as a function call. (The call overhead likely exceeds the cost of the loop.) Nonetheless, the example works well for explaining the principles of syntax-driven computation.

```
int stoi( char *s ) {
  int i = 0;
  while('0' ≤ *s ≤ '9') {
    i = i * 10
      + ((int)*s - '0');
    s++;
  }
  return i;
}
```

An Equivalent Treewalk Formulation

These integer-grammar value computations can also be written as recursive treewalks over syntax trees. Fig. 5.3(a) shows the syntax tree for "175" with



(a) Syntax Tree for "175" with the Left-recursive Grammar

```
Value(root)
    if (root is type Number) then
        return Value(son(root))
    else if (root is type DList) then
        return 10 × Value(left(root))
                + Value(right(root))
    else if (root is type digit) then
        return integer(root's lexeme)
```

(b) Treewalk to Compute the Value from the Syntax Tree

■ **FIGURE 5.3** Treewalk Computations for the Positive Integer Grammar.

the left recursive grammar. Panel (b) shows a simple treewalk to compute its value. It uses "*integer(c)*" to convert a single character to an integer value.

The treewalk formulation exposes several important aspects of yacc-style syntax-driven computation. Information flows up the syntax tree from the leaves toward the root. The action associated with a production only has names for values associated with grammar symbols named in the production. Bottom-up information flow works well in this paradigm. Top-down information flow does not.

The restriction to bottom-up information flow might appear problematic. In fact, the compiler writer can reach around the paradigm and evade these restrictions by using nonlocal variables and data structures in the "actions." Indeed, one use for a compiler's symbol table is precisely to provide nonlocal access to data derived by syntax-driven computations.

In principle, any top-down information flow problem can be solved with a bottom-up framework by passing all of the information upward in the tree to a common ancestor and solving the problem at that point. In practice, that idea does not work well because (1) the implementor must plan all the information flow; (2) she must write code to implement it; and (3) the computed result appears at a point in the tree far from where it is needed. In practice, it is often better to rethink the computation than to pass all of that information around the tree.

Form of the Grammar

Because the grammar dictates the sequence of actions, its shape affects the computational strategy. Consider a right-recursive version of the grammar for positive integers. It reduces the rightmost digit first, which suggests the following approach:

```
Number  : DList         { return second($1); } ;
DList   : digit DList   { $$ = pair(10 * first($2),
                              first($2) * CToI($1) + second($2)); }
        | digit         { $$ = pair(10,CToI($1)); } ;
```

This scheme accumulates, right to left, both a multiplier and a value. To store both values with a *DList*, it uses a `pair` constructor and the functions `first` and `second` to access a pair's component values. While this paradigm works, it is much harder to understand than the mechanism for the left-recursive grammar.

In grammar design, the compiler writer should consider the kinds of computation that she wants the parser to perform. Sometimes, changing the grammar can produce a simpler, faster computation.

5.3.2 Translating Expressions

Expressions form a large portion of most programs. If we consider them as trees—that is, trees rather than directed acyclic graphs—then they are a natural example for syntax-driven translation. Fig. 5.4 shows a simple syntax-driven framework to build an abstract syntax tree for expressions. The rules are simple.

- If a production contains an operator, it builds an interior node to represent the operator.
- If a production derives a name or number, it builds a leaf node and records the lexeme.

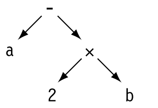
Production			Syntax-Driven Actions
Expr	→	*Expr* + *Term*	{ $$ ← *MakeNode2*(`plus`,$1,$3); };
	|	*Expr* − *Term*	{ $$ ← *MakeNode2*(`minus`,$1,$3); };
	|	*Term*	{ $$ ← $1; };
Term	→	*Term* × *Factor*	{ $$ ← *MakeNode2*(`times`,$1,$3); };
	|	*Term* ÷ *Factor*	{ $$ ← *MakeNode2*(`divide`,$1,$3); };
	|	*Factor*	{ $$ ← $1; };
Factor	→	(*Expr*)	{ $$ ← $2; };
	|	`number`	{ $$ ← *MakeLeaf*(`number`,*lexeme*); };
	|	`name`	{ $$ ← *MakeLeaf*(`name`,*lexeme*); };

■ **FIGURE 5.4** Building an Abstract Syntax Tree.

Production		Syntax-Driven Actions
Expr →	*Expr* + *Term*	{ $$ ← *NextRegister*() ; *Emit*(add, $1, $3, $$) ; };
|	*Expr* − *Term*	{ $$ ← *NextRegister*() ; *Emit*(sub, $1, $3, $$) ; };
|	*Term*	{ $$ ← $1 ; };
Term →	*Term* × *Factor*	{ $$ ← *NextRegister*() ; *Emit*(mult, $1, $3, $$) ; };
|	*Term* ÷ *Factor*	{ $$ ← *NextRegister*() ; *Emit*(div, $1, $3, $$) ; };
|	*Factor*	{ $$ ← $1 ; };
Factor →	(*Expr*)	{ $$ ← $2 ; };
|	number	{ $$ ← *NumberIntoReg*(*lexeme*) ; };
|	name	{ entry ← *STLookup*(*lexeme*); $$ ← *ValueIntoReg*(entry) ; };

■ **FIGURE 5.5** Emitting Three-Address Code for Expressions.

- If the production exists to enforce precedence, it passes the AST for the subexpression upward.



An AST for a - 2 × b

The code uses two constructors to build the nodes. *MakeNode2(a, b, c)* builds a binary node of type *a* with children *b* and *c*. *MakeLeaf(*name*, a)* builds a leaf node and associates it with the lexeme *a*. For the expression a - 2 × b, this translation scheme would build the simple AST shown in the margin.

ASTs have a direct and obvious relationship to the grammatical structure of the input program. Three-address code lacks that direct mapping. Nonetheless, a syntax-driven framework can easily emit three-address code for expressions and assignments. Fig. 5.5 shows a syntax-driven framework to emit ILOC-like code from the classic expression grammar. The framework assumes that values reside in memory at the start of the expression.

To simplify the framework, the compiler writer has provided high-level functions to abstract away the details of where values are stored.

- *NextRegister* returns a new register number.
- *NumberIntoReg* returns the number of a register that holds the constant value from the lexeme.
- *STLookup* takes a name as input and returns the symbol table entry for the entity to which the name is currently bound.

- *ValueIntoReg* returns the number of a register that holds the current value of the name from the lexeme.

If the grammar included assignment, it would need a helper function *RegIntoMemory* to move a value from a register into memory.

Helper functions such *NumberIntoReg* and *ValueIntoReg* must emit three-address code that represents the access methods for the named entities. If the IR only has low-level operations, as occurs in ILOC, these functions can become complex. The alternative approach is to introduce high-level operations into the three-address code that preserve the essential information, and to defer elaboration of these operations until after the compiler fully understands the storage map and the access methods.

Most of the functions have obvious implementations. For example, *NumberIntoReg*('2') can emit a load immediate operation.

Applying this syntax-driven translation scheme to the expression a - 2 × b produces the ILOC code shown in the margin. The code assumes that r_{arp} holds a pointer to the procedure's local data area and that @a and @b are the offsets from r_{arp} at which the program stores the values of a and b. The code leaves the result in r_5.

```
loadAI  r_arp, @a  ⇒ r_1
loadI   2          ⇒ r_2
loadAI  r_arp, @b  ⇒ r_3
mult    r_2, r_3   ⇒ r_4
sub     r_1, r_4   ⇒ r_5
```

ILOC Code for a - 2 × b

Implementation in an LR(1) Parser

This style of syntax-driven computation was introduced in yacc, an early LALR(1) parser generator. The implementation requires two changes to the LR(1) skeleton parser. Understanding those changes sheds insight on both the yacc notation and how to use it effectively. Fig. 5.6 shows the modified skeleton LR(1) parser. Changes are typeset in ***bold typeface***.

The first change creates storage for the value associated with a grammar symbol in the derivation. The original skeleton parser stored its state in $\langle symbol, state \rangle$ pairs kept on a stack, where *symbol* was a grammar symbol and *state* was a parser state. The modified parser replaces those pairs with $\langle symbol, state, value \rangle$ triples, where *value* holds the entity assigned to $$ in the reduction that shifted the triple onto the stack. Shift actions use the value of the lexeme.

Parser generators differ in what value they assign to a terminal symbol.

The second change causes the parser to invoke a function called *PerformActions* before it reduces. The parser uses the result of that call in the value field when it pushes the new triple onto the stack.

The parser generator constructs *PerformActions* from the translation actions specified for each production in the grammar. The skeleton parser passes the function a production number; the function consists of a case statement that switches on that production number to the appropriate snippet of code for the reduction.

```
push ⟨INVALID, INVALID, INVALID⟩ onto the stack
push ⟨start symbol, $s_0$, INVALID⟩ onto the stack
word ← NextWord()
while (true) do
    state ← state from triple at top of stack
    if Action[state,word] = "reduce $A \rightarrow \beta$" then
        value ← PerformActions($A \rightarrow \beta$)
        pop $|\beta|$ triples from the stack
        state ← state from triple at top of stack
        push ⟨A, Goto[state, A], value⟩ onto the stack
    else if Action[state,word] = "shift $s_i$" then
        push ⟨word, $s_i$, lexeme ⟩ onto the stack
        word ← NextWord()
    else if Action[state,word] = "accept" and word = eof
        then break
    else throw a syntax error
report success /* executed the "accept" case */
```

■ **FIGURE 5.6** The Skeleton LR(1) Parser with Translation Support.

The remaining detail is to translate the `yacc`-notation symbols `$$`, `$1`, `$2`, and so on into concrete references into the stack. `$$` represents the return value for *PerformActions*. Any other symbol, `$`*i*, is a reference to the value field of the triple corresponding to symbol *i* in the production's RHS. Since those triples appear, in right to left order, on the top of the stack, `$`*i* translates to the value field for the triple located $|\beta| - i$ slots from the top of the stack.

Handling Nonlocal Computation

The examples so far only show local computation in the grammar. Individual rules can only name symbols in the same production. Many of the tasks in a compiler require information from other parts of the computation; in a treewalk formulation, they need data from distant parts of the syntax tree.

Defining occurrence
The first occurrence of a name in a given scope is its defining occurrence.

Any subsequent use is a *reference occurrence*.

One example of nonlocal computation in a compiler is the association of type, lifetime, and visibility information with a named entity, such as a variable, procedure, object, or structure layout. The compiler becomes aware of the entity when it encounters the name for the first time in a scope—the name's *defining occurrence*. At the defining occurrence of a name x, the compiler must determine x's properties. At subsequent reference occurrences, the compiler needs access to those previously determined properties.

The kinds of rules introduced in the previous example provide a natural mechanism to pass information up the parse tree and to perform local computation—between values associated with a node and its children. To translate an expression such as `x + y` into a low-level three-address IR, the compiler must access information that is distant in the parse tree—the declarations of `x` and `y`. If the compiler tries to generate low-level three-address code for the expression, it may also need access to information derived from the syntax, such as a determination as to whether or not the code can keep `x` in a register—that is, whether or not `x` is ambiguous. A common way to address this problem is to store information needed for nonlocal computations in a globally accessible data structure. Most compilers use a symbol table for this purpose (see Section 4.5).

The use of a global symbol table to provide nonlocal access is analogous to the use of global variables in imperative programs.

The "symbol table" is actually a complex set of tables and search paths. Conceptually, the reader can think of it as a hashmap tailored to each scope. In a specific scope, the search path consists of an ordered list of tables that the compiler will search to resolve a name.

Different parts of the grammar will manipulate the symbol table representation. A name's defining occurrence creates its symbol table entry. Its declarations, if present, set various attributes and bindings. Each reference occurrence will query the table to determine the name's attributes. Statements that open a new scope, such as a procedure, a block, or a structure declaration, will create new scopes and link them into the search path. More subtle issues may arise; if a C program takes the address of a variable `a`, as in `&a`, the compiler should mark `a` as potentially ambiguous.

In a dynamically typed language such as PYTHON, statements that define *x* may change *x*'s attributes.

The same trick, using a global variable to communicate information between the translation rules, arises in other contexts. Consider a source language with a simple declaration syntax. The parser can create symbol-table entries for each name and record their attributes as it processes the declarations. For example, the source language might include syntax similar to the following set of rules:

```
Declaration  →  TypeSpec NameList  { CurType ← invalid; };
TypeSpec     →  int                { CurType ← int; };
             |  float              { CurType ← float; };
NameList     →  NameList , name    { err ← SetType($2,CurType); };
             |  name               { err ← SetType($1,CurType); };
```

where `SetType` creates a new entry for `name` if none exists and reports an error if `name` exists and has a designated type other than `CurType`.

The type of the declared variables is determined in the productions for *TypeSpec*. The action for *TypeSpec* records the type into a global variable, `CurType`. When a name appears in the *NameList* production, the action sets the `name`'s type to the value in `CurType`. The compiler writer has reached

SINGLE-PASS COMPILERS

In the earliest days of compilation, implementors tried to build single-pass compilers—translators that would emit assembly code or machine code in a single pass over the source program. At a time when fast computers were measured in kiloflops, the efficiency of translation was an important issue.

To simplify single-pass translation, language designers adopted rules meant to avoid the need for multiple passes. For example, PASCAL requires that all declarations occur before any executable statement; this restriction allowed the compiler to resolve names and perform storage layout before emitting any code. In hindsight, it is unclear whether these restrictions were either necessary or desirable.

Making multiple passes over the code allows the compiler to gather more information and, in many cases, to generate more efficient code, as Floyd observed in 1961 [160]. With today's more complex processors, almost all compilers perform multiple passes over an IR form of the code.

around the paradigm to pass information from the RHS of one production to the RHS of another.

Form of the Grammar

The form of the grammar can play an important role in shaping the computation. To avoid the global variable CurType in the preceding example, the compiler writer might reformulate the grammar for declaration syntax as follows:

```
Declaration  →  int INameList
             |  float FNameList

INameList    →  NameList name    { err ← SetType($2, int); };
             |  name             { err ← SetType($1, int); };

FNameList    →  NameList name    { err ← SetType($2, float); };
             |  name             { err ← SetType($1, float); };
```

This form of the grammar accepts the same language. However, it creates distinct name lists for int and float names, As shown, the compiler writer can use these distinct productions to encode the type directly into the syntax-directed action. This strategy simplifies the translation framework and eliminates the use of a global variable to pass information between the productions. The framework is easier to write, easier to understand, and, likely, easier to maintain. Sometimes, shaping the grammar to the computation can simplify the syntax-driven actions.

Tailoring Expressions to Context

A more subtle form of nonlocal computation can arise when the compiler writer needs to make a decision based on information in multiple productions. For example, consider the problem of extending the framework in Fig. 5.5 so that it can emit an immediate multiply operation (multI in ILOC) when translating an expression. In a single-pass compiler, for example, it might be important to emit the multI in the initial IR.

For the expression a × 2, the framework in Fig. 5.5 would produce something similar to the code shown in the margin. (The code assumes that a resides in r_a.) The reduction by *Factor* → number emits the loadI; it executes before the reduction by *Term* → *Term* × *Factor*.

loadI 2 ⇒ r_i
mult r_a, r_i ⇒ r_j

ILOC Code for a × 2

To recognize the opportunity for a multI, the compiler writer would need to add code to the action for *Term* → *Term* × *Factor* that recognizes when $3 contains a small integer constant and generates the multI in that case. The commutative case would require a similar check on $1. Even with this effort, the loadI would remain. Subsequent optimization could remove it (see Section 10.2).

The fundamental problem is that the actions in our syntax-driven translation can only access local information because they can only name symbols in the current production. That structure forces the translation to emit the loadI before it can know that the value's use occurs in an operation that has an "immediate" variant.

The obvious suggestion is to refactor the grammar to reflect the multI case. If the compiler writer rewrites *Term* → *Term* × *Factor* with the three productions shown in the margin, then she can emit a multI in the action for *Term* → *Term* × number, which will catch the case a × 2. It will not, however, catch the case 2 × a. Forward substitution on the left operand will not work, because the grammar is left recursive. At best, forward substitution can expose either an immediate left operand or an immediate right operand.

Term → *Term* × (*Expr*)
| *Term* × number
| *Term* × name

The most comprehensive solution to this problem is to create the more general multiply operation and allow either subsequent optimization or instruction selection to discover the opportunity and rewrite the code. Either of the techniques for instruction selection described in Chapter 11 can discover the opportunity for multI and rewrite the code accordingly.

If the compiler must generate the multI early, the most rational approach is to have the compiler maintain a small buffer of three to four operations and to perform *peephole optimization* as it emits the initial IR (see Section 11.3.1). It can easily detect and rewrite inefficiencies such as this one.

Peephole optimization
an optimization that applies pattern matching to simplify code in a small buffer

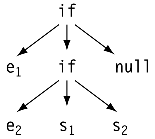
1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *WithElse* `else` *Stmt*
3		|	*Other*
4	*WithElse*	→	`if` *Expr* `then` *WithElse* `else` *WithElse*
5		|	*Other*

■ **FIGURE 5.7** The Unambiguous If-Then-Else Grammar.

5.3.3 Translating Control-Flow Statements

As we have seen, the IR for expressions follows closely from the syntax for expressions, which leads to straightforward translation schemes. Control-flow statements, such as nested `if–then–else` constructs or loops, can require more complex representations.

Building an AST



AST for Nested If Statements

The parser can build an AST to represent control-flow constructs in a natural way. Consider a nest of `if–then–else` constructs, using the grammar from Fig. 5.7. The AST can use a node with three children to represent the `if`. One child holds the control expression; another holds the statements in the `then` clause; the third holds the statements in the `else` clause. The drawing in the margin shows the AST for the input:

```
if e1 then if e2 then s1 else s2
```

The actions to build this AST are straightforward.

Building Three-Address Code

To translate an `if–then–else` construct into three-address code, the compiler must encode the transfers of control into a set of labels, branches, and jumps. The three-address IR resembles the obvious assembly code for the construct:

1. evaluate the control expression;
2. branch to the `then` subpart (s_1) or the `else` subpart (s_2) as appropriate;
3. at the end of the selected subpart, jump to the start of the statement that follows the `if–then–else` construct—the "exit."

This translation scheme requires labels for the `then` part, the `else` part, and the exit, along with a branch and two jumps.

Production 4 in the grammar from Fig. 5.7 shows the issues that arise in a translation scheme to emit ILOC-like code for an `if–then–else`. Other productions will generate the IR to evaluate the *Expr* and to implement the

```
WithElse      →  if Expr CreateBranch
                    then WithElse ToExit1
                    else WithElse ToExit2

CreateBranch  →  ϵ
ToExit1       →  ϵ
ToExit2       →  ϵ
```

■ **FIGURE 5.8** Creating Mid-Production Actions.

`then` and `else` parts. The scheme for rule 4 must combine these disjoint parts into code for a complete `if–then–else`.

The complication with rule 4 lies in the fact that the parser needs to emit IR at several different points: after the *Expr* has been recognized, after the *WithElse* in the `then` part has been recognized, and after the *WithElse* in the `else` part has been recognized. In a straightforward rule set, the action for rule 4 would execute after all three of those subparts have been parsed and the IR for their evaluation has been created.

The scheme for rule 4 must have several different actions, triggered at different points in the rule. To accomplish this goal, the compiler writer can modify the grammar in a way that creates reductions at the points in the parse where the translation scheme needs to perform some action.

Fig. 5.8 shows a rewritten version of production 4 that creates reductions at the critical points in the parse of a nested `if–then–else` construct. It introduces three new nonterminal symbols, each defined by an epsilon production.

The reduction for *CreateBranch* can create the three labels, insert the conditional branch, and insert a `nop` with the label for the `then` part. The reduction for *ToExit1* inserts a jump to the exit label followed by a `nop` with the label for the `else` part. Finally, *ToExit2* inserts a jump to the exit label followed by a `nop` with the exit label.

The compiler could omit the code for *ToExit2* and rely on the fall-through case of the branch.

Making the branch explicit rather than implicit gives later passes more freedom to reorder the code (see Section 8.6.2).

One final complication arises. The compiler writer must account for nested constructs. The three labels must be stored in a way that both ties them to this specific instance of a *WithElse* and makes them accessible to the other actions associated with rule 4. Our notation, so far, does not provide a solution to this problem. The `bison` parser generator extended `yacc` notation to solve it, so that the compiler writer does not need to introduce an explicit stack of label-valued triples.

The `bison` solution is to allow an action between any two symbols on the production's RHS. It behaves as if `bison` inserts a new nonterminal at the point of the action, along with an ϵ-production for the new nonterminal. It then associates the action with this new ϵ-production. The compiler writer must count carefully; the presence of a mid-production action creates an extra name and increments the names of symbols to its right.

Using this scheme, the mid-production actions can access the stack slot associated with any symbol in the expanded production, including the symbol on the LHS of rule 4. In the `if–then–else` scenario, the action between *Expr* and `then` can store a triple of labels temporarily in the stack slot for that LHS, `$$`. The actions that follow the two *WithElse* clauses can then find the labels that they need in `$$`. The result is not elegant, but it creates a workaround to allow slightly nonlocal access.

Case statements and loops present similar problems. The compiler needs to encode the control-flow of the original construct into a set of labels, branches, and jumps. The parse stack provides a natural way to keep track of the information for nested control-flow structures.

SECTION REVIEW

As part of translation, the compiler produces an IR form of the code. To support that initial translation, parser generators provide a facility to specify syntax-driven computations that tie computation to the underlying grammar. The parser then sequences those actions based on the actual syntax of the input program.

Syntax-driven translation creates an efficient mechanism for IR generation. It easily accommodates decisions that require either local knowledge or knowledge from earlier in the parse. It cannot make decisions based on facts that appear later in the parse. Such decisions require multiple passes over the IR to refine and improve the code.

REVIEW QUESTIONS

1. The grammar in the margin defines the syntax of a simple four-function calculator. The calculator displays its current result on each reduction to *Expr* or *Term*. Write the actions for a syntax-driven scheme to evaluate expressions with this grammar.
2. Consider the grammar from Fig. 5.7. Write a set of translation rules to build an AST for an `if-then-else` construct.

$$
\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
 & | & \mathit{Expr} - \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \times \texttt{number} \\
 & | & \mathit{Term} \div \texttt{number} \\
 & | & \texttt{number}
\end{array}
$$

Simple Calculator Grammar

5.4 MODELING THE NAMING ENVIRONMENT

Modern programming languages allow the programmer to create complex name spaces. Most languages support some variant of a lexical naming hierarchy, where visibility, type, and lifetime are expressed in relationship to the structure of the program. Many languages also support an object-oriented naming hierarchy, where visibility and type are relative to inheritance and lifetimes are governed by explicit or implicit allocation and deallocation. During translation, optimization, and code generation, the compiler needs mechanisms to model and understand these hierarchies.

When the compiler encounters a name, its syntax-driven translation rules must map that name to a specific entity, such as a variable, object, or procedure. That name-to-entity binding plays a key role in translation, as it establishes the name's type and access method, which, in turn, govern the code that the compiler can generate. The compiler uses its model of the name space to determine this binding—a process called *name resolution*.

A program's name space can contain multiple subspaces, or scopes. As defined in Chapter 4, a scope is a region in the program that demarcates a name space. Inside a scope, the programmer can define new names. Names are visible inside their scope and, generally, invisible outside their scope.

Static binding
When the compiler can determine the name-to-entity binding, we consider that binding to be static, in that it does not change at runtime.

Dynamic binding
When the compiler cannot determine the name-to-entity binding and must defer that resolution until runtime, we consider that binding to be dynamic.

The primary mechanism used to model the naming environment is a set of tables, collectively referred to as the symbol table. The compiler builds these tables during the initial translation. For names that are bound statically, it annotates references to the name with a specific symbol table reference. For names that are bound dynamically, such as a C++ virtual function, it must make provision to resolve that binding at runtime. As the parse proceeds, the compiler creates, modifies, and discards parts of this model.

Before discussing the mechanism to build and maintain the visibility model, a brief review of scope rules is in order.

5.4.1 Lexical Hierarchies

Most programming languages provide nested lexical scopes in some form. The general principle behind lexical scope rules is simple:

Lexical scope
Scopes that nest in the order that they are encountered in the program are often called *lexical scopes*.

> *At a point* p *in a program, an occurrence of name* n *refers to the entity named* n *that was created, explicitly or implicitly, in the scope that is lexically closest to* p.

Thus, if *n* is used in the current scope, it refers to the *n* declared in the current scope, if one exists. If not, it refers to the declaration of *n* that occurs in the

CREATING A NEW NAME

Programming languages differ in the way that the programmer declares names. Some languages require a declaration for each named variable and procedure. Others determine the attributes of a name by applying rules in place at the name's defining occurrence. Still others rely on context to infer the name's attributes.

The treatment of a defining occurrence of some name *x* in scope *S* depends on the source language's visibility rules and the surrounding context.

- If *x* occurs in a declaration statement, then the attributes of *x* in *S* are obvious and well-defined.
- If *x* occurs as a reference and an instance of *x* is visible in a scope that surrounds *S*, most languages bind *x* to that entity.
- If *x* occurs as a reference and no instance of *x* is visible, then treatment varies by language. APL, PYTHON and even FORTRAN create a new entity. C treats the reference as an error.

When the compiler encounters a defining occurrence, it must create the appropriate structures to record the name and its attributes and to make the name visible to name lookups.

closest enclosing scope. The outermost scope typically contains names that are visible throughout the entire program, usually called *global* names.

Programming languages differ in the ways that they demarcate scopes. PASCAL marks a scope with a `begin`–`end` pair. C defines a scope between each pair of curly braces, { and }. Structure and record definitions create a scope that contains their element names. Class definitions in an OOL create a new scope for names visible in the class.

PASCAL uses curly braces as the comment delimiter.

To make the discussion concrete, consider the PASCAL program shown in Fig. 5.9. It contains five distinct scopes, one for each procedure: `Main`, `Fee`, `Fie`, `Foe`, and `Fum`. Each procedure declares some variables drawn from the set of names `x`, `y`, and `z`. In the code, each name has a subscript to indicate its level number. Names declared in a procedure always have a level that is one more than the level of the procedure name. Thus, `Main` has level 0, and the names `x`, `y`, `z`, `Fee`, and `Fie`, all declared directly in `Main`, have level 1.

Static coordinate
For a name `x` declared in scope *s*, its static coordinate is a pair ⟨*l*,*o*⟩ where *l* is the lexical nesting level of *s* and *o* is the offset where `x` is stored in the scope's data area.

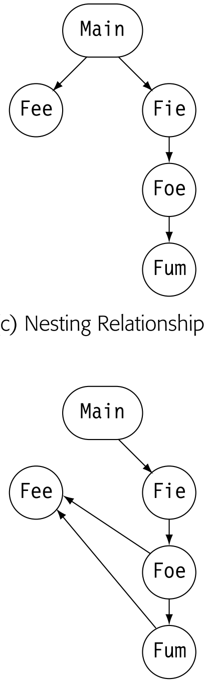
To represent names in a lexically scoped language, the compiler can use the *static coordinate* for each name. The static coordinate is a pair ⟨*l*,*o*⟩, where *l* is the name's lexical nesting level and *o* is the its offset in the data area for level *l*. The compiler obtains the static coordinate as part of the process of name resolution—mapping the name to a specific entity.

```
program Main0(input0, output0);
  var x1, y1, z1: integer;
  procedure Fee1;
     var x2: integer;
     begin { Fee1 }
        x2 := 1;
        y1 := x2 * 2 + 1
     end;
  procedure Fie1;
     var y2: real;
     procedure Foe2;
        var z3: real;
           procedure Fum3;
              var y4: real;
              begin { Fum3 }
                 x1 := 1.25 * z3;
                 Fee1;
                 writeln('x = ',x1)
              end;
        begin { Foe2 }
           z3 := 1;
           Fee1;
           Fum3
        end;
     begin { Fie1 }
        Foe2;
        writeln('x = ',x1)
     end;
  begin { Main0 }
     x1 := 0;
     Fie1
  end.
```

(a) PASCAL Program

Scope	x	y	z
Main	⟨1, 0⟩	⟨1, 4⟩	⟨1, 8⟩
Fee	⟨2, 0⟩	⟨1, 4⟩	⟨1, 8⟩
Fie	⟨1, 0⟩	⟨2, 0⟩	⟨1, 8⟩
Foe	⟨1, 0⟩	⟨2, 0⟩	⟨3, 0⟩
Fum	⟨1, 0⟩	⟨4, 0⟩	⟨3, 0⟩

(b) Static Coordinates



(c) Nesting Relationships

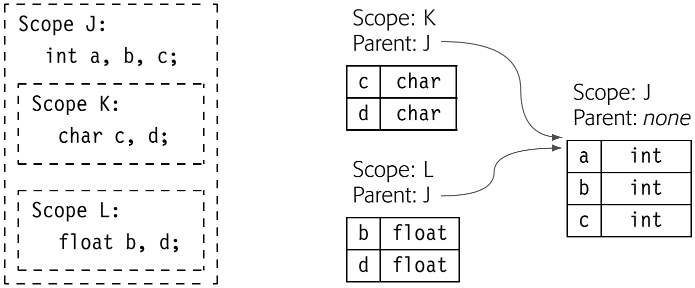
(d) Calling Relationships

■ **FIGURE 5.9** Nested Lexical Scopes in PASCAL.

Modeling Lexical Scopes

As the parser works its way through the input code, it must build and maintain a model of the naming environment. The model changes as the parser enters and leaves individual scopes. The compiler's symbol table instantiates that model.

The compiler can build a separate table for each scope, as shown in Fig. 5.10. Panel (a) shows an outer scope J that contains two inner scopes.



(a) Nested Scopes

(b) Corresponding Scope Tables

■ **FIGURE 5.10** Tables for the Lexical Hierarchy.

In scope `K`, `a` and `b` have type `int` while `c` and `d` have type `char`. In scope `L`, `a` and `c` have type `int` while `b` and `d` have type `float`.

Scopes `K` and `L` are both nested inside scope `J`. Scopes `K` and `L` are otherwise unrelated.

Panel (b) shows the corresponding symbol tables. The table for a scope consists of both a hash table and a link to the surrounding scope. The gray arrows depict the search path, which reflects nesting in the code. Thus, a lookup of `a` in scope `K` would fail in the table for `K`, then follow the link to scope `J`, where it would find the definition of `a` as an `int`.

This approach lets the compiler create flexible, searchable models for the naming environment in each scope. A *search path* is just a list or chain of tables that specifies the order in which the tables will be searched. At compile time, a lookup for name resolution begins with the search path for the current scope and proceeds up the chain of surrounding scopes. Because the relationship between scopes is static (unchanging), the compiler can build scope-specific search paths with syntax-driven translation and preserve those tables and paths for use in later stages of the compiler and, if needed, in other tools.

Building the Model

The compiler writer can arrange to build the name-space model during syntax-driven translation. The source language constructs that enter and leave distinct scopes can trigger actions to create tables and search paths. The productions for declarations and references can create and refine the entries for names.

- *Block Demarcations* such as `begin` and `end`, { and }, and procedure entry and exit, create a new table on entry to the scope and link it to the start of the search path for the block(s) associated with the current scope. On exit, the action should mark the table as final.

DYNAMIC SCOPING

The alternative to lexical scoping is dynamic scoping. The distinction between lexical and dynamic scoping only matters when a procedure refers to a variable that is declared outside the procedure's own scope, sometimes called a *free variable*.

With lexical scoping, the rule is simple and consistent: a free variable is bound to the declaration for its name that is lexically closest to the use. If the compiler starts in the scope containing the use, and checks successive surrounding scopes, the variable is bound to the first declaration that it finds. The declaration always comes from a scope that encloses the reference.

With dynamic scoping, the rule is equally simple: a free variable is bound to the variable by that name that was most recently created at runtime. Thus, when execution encounters a free variable, it binds that free variable to the most recent instance of that name. Early implementations created a runtime stack of names on which every name was pushed as its defining occurrence was encountered. To bind a free variable, the running code searched the name stack from its top downward until a variable with the right name was found. Later implementations are more efficient.

While many early Lisp systems used dynamic scoping, lexical scoping has become the dominant choice. Dynamic scoping is easy to implement in an interpreter and somewhat harder to implement efficiently in a compiler. It can create bugs that are difficult to detect and hard to understand. Dynamic scoping still appears in some languages; for example, Common Lisp still allows the program to specify dynamic scoping.

- *Variable Declarations,* if they exist, create entries for the declared names in the local table and populate them with the declared attributes. If they do not exist, then attributes such as type must be inferred from references. Some size information might be inferred from points where aggregates are allocated.
- *References* trigger a lookup along the search path for the current scope. In a language with declarations, failure to find a name in the local table causes a search through the entire search path. In a language without declarations, the reference may create a local entity with that name; it may refer to a name in a surrounding scope. The rules on implicit declarations are language specific.

  FORTRAN creates the name with default attributes based on the first letter of the name. C looks for it in surrounding scopes and declares an error if it is not found. PYTHON's actions depend on whether the first occurrence of the name in a scope is a definition or a use.

Examples

Lexical scope rules are generally similar across different programming languages. However, language designers manage to insert surprising and idiosyncratic touches. The compiler writer must adapt the general translation schemes described here to the specific rules of the source language.

C has a simple, lexically scoped name space. Procedure names and global variables exist in the global scope. Each procedure creates its own local scope for variables, parameters, and labels. C does not include nested procedures or functions, although some compilers, such as GCC, implement this extension. Blocks, set off by { and }, create their own local scopes; blocks can be nested.

The C keyword `static` both restricts a name's visibility and specifies its lifetime. A `static` global name is only visible inside the file that contains its declaration. A `static` local name has local visibility. Any `static` name has a global lifetime; that is, it retains its value across distinct invocations of the declaring procedure.

SCHEME has scope rules that are similar to those in C. Almost all entities in SCHEME reside in a single global scope. Entities can be data; they can be executable expressions. System-provided functions, such as `cons`, live alongside user-written code and data items. Code, which consists of an executable expression, can create private objects by using a `let` expression. Nesting `let` expressions inside one another can create nested lexical scopes of arbitrary depth.

PYTHON is an Algol-like language that eschews declarations. It supports three kinds of scopes: a local function-specific scope for names defined in a function; a global scope for names defined outside of any programmer-supplied function; and a builtin scope for implementation-provided names such as `print`. These scopes nest in that order: local embeds in global which embeds in builtin. Functions themselves can nest, creating a hierarchy of local scopes.

PYTHON does not provide type declarations. The first use of a name *x* in a scope is its defining occurrence. If the first use assigns *x* a value, then it binds *x* to a new local entity with its type defined by the assigned value. If the first use refers to *x*'s value, then it binds *x* to a global entity; if no such entity exists, then that defining use creates the entity. If the programmer intends *x* to be global but needs to define it before using it, the programmer can add a `nonlocal` declaration for the name, which ensures that *x* is in the global scope.

TERMINOLOGY FOR OBJECT-ORIENTED LANGUAGES

The diversity of object-oriented languages has led to some ambiguity in the terms that we use to discuss them. To make the discussion in this chapter concrete, we will use the following terms:

Object An object is an abstraction with one or more members. Those members can be data items, code that manipulates those data items, or other objects. An object with code members is a *class*. Each object has internal state—data whose lifetimes match the object's lifetime.

Class A class is a collection of objects that all have the same abstract structure and characteristics. A class defines the set of data members in each *instance* of the class and defines the code members, or *methods*, that are local to that class. Some methods are *public*, or externally visible, while others are *private*, or invisible outside the class.

Inheritance Inheritance is a relationship among classes that defines a partial order on the name scopes of classes. A class a may inherit members from its *superclass*. If a is the superclass of b, b is a *subclass* of a. A name x defined in a subclass obscures any definitions of x in a superclass. Some languages allow a class to inherit from multiple superclasses.

Receiver Methods are invoked relative to some object, called the method's receiver. The receiver is known by a designated name inside the method, such as `this` or `self`.

The power of an OOL arises, in large part, from the organizational possibilities presented by its multiple name spaces.

5.4.2 Inheritance Hierarchies

Object-oriented languages (OOLs) introduce a second form of nested name space through inheritance. OOLs introduce *classes*. A class consists of a collection (possibly empty) of objects that have the same structure and behavior. The class definition specifies the code and data members of an object in the class.

Much of the power of an OOL derives from the ability to create new classes by drawing on definitions of other existing classes. In JAVA terminology, a new class β can extend an existing class α; objects of class β then inherit definitions of code and data members from the definition of α. β may redefine names from α with new meanings and types; the new definitions obscure earlier definitions in α or its superclasses. Other languages provide similar functionality with a different vocabulary.

Polymorphism
The ability of an entity to take on different types is often called *polymorphism*.

The terminology used to specify inheritance varies across languages. In JAVA, a subclass *extends* its superclass. In C++, a subclass is *derived* from its superclass.

Subtype polymorphism
the ability of a subclass object to reference superclass members

Class extension creates an *inheritance hierarchy*: if α is the *superclass* of β, then any method defined in α must operate correctly on an object of class β, provided that the method is visible in β. The converse is not true. A subclass method may rely on subclass members that are not defined in instances of the superclass; such a method cannot operate correctly on an object that is an instance of the superclass.

In a single-inheritance language, such as JAVA, inheritance imposes a tree-structured hierarchy on the set of classes. Other languages allow a class β to have multiple immediate superclasses. This notion of "multiple inheritance" gives the programmer an ability to reuse more code, but it creates a more complex name resolution environment.

Each class definition creates a new scope. Names of code and data members are specific to the class definition. Many languages provide an explicit mechanism to control the visibility of member names. In some languages, class definitions can contain other classes to create an internal lexical hierarchy. Inheritance defines a second search path based on the superclass relationship.

In translation, the compiler must map an $\langle object, member \rangle$ pair back to a specific member declaration in a specific class definition. That binding provides the compiler with the type information and access method that it needs to translate the reference. The compiler finds the object name in the lexical hierarchy; that entry provides a class that serves as the starting point for the compiler to search for the member name in the inheritance hierarchy.

Modeling Inheritance Hierarchies

The lexical hierarchy reflects nesting in the syntax. The inheritance hierarchy is created by definitions, not syntactic position.

To resolve member names, the compiler needs a model of the inheritance hierarchy as defined by the set of class declarations. The compiler can build a distinct table for the scope associated with each class as it parses that class' declaration. Source-language phrases that establish inheritance cause the compiler to link class scopes together to form the hierarchy. In a single-inheritance language, the hierarchy has a tree structure; classes are children of their superclasses. In a multiple-inheritance language, the hierarchy forms an acyclic graph.

The compiler uses the same tools to model the inheritance hierarchy that it does to model the lexical hierarchy. It creates tables to model each scope. It links those tables together to create search paths. The order in which those searches occur depends on language-specific scope and inheritance rules. The underlying technology used to create and maintain the model does not.

Compile-Time Versus Runtime Resolution

The major complication that arises with some OOLs derives not from the presence of an inheritance hierarchy, but rather from when that hierarchy is defined. If the OOL requires that class definitions be present at compile time and that those definitions cannot change, then the compiler can resolve member names, perform appropriate type checking, determine appropriate access methods, and generate code for member-name references. We say that such a language has a *closed class structure*.

Closed class structure
If the class structure of an application is fixed at compile time, the OOL has a *closed hierarchy*.

By contrast, if the language allows the running program to change its class structure, either by importing class definitions at runtime, as in JAVA, or by editing class definitions, as in SMALLTALK, then the language may need to defer some name resolution and binding to runtime. We say that such a language has an *open class structure*.

Open class structure
If an application can change its class structure at runtime, it has an *open hierarchy*.

Lookup with Inheritance

Assume, for the moment, a closed class structure. Consider two distinct scenarios:

1. If the compiler finds a reference to an unqualified name *n* in some procedure *p*, it searches the lexical hierarchy for *n*. If *p* is a method defined in some class *c*, then *n* might also be a data member of *c* or some superclass of *c*; thus, the compiler must insert part of the inheritance hierarchy into the appropriate point in the search path.
2. If the compiler finds a reference to member *m* of object *o*, it first resolves *o* in the lexical hierarchy to an instance of some class *c*. Next, it searches for *m* in the table for class *c*; if that search fails, it looks for *m* in each table along *c*'s chain of superclasses (in order). It either finds *m* or exhausts the hierarchy.

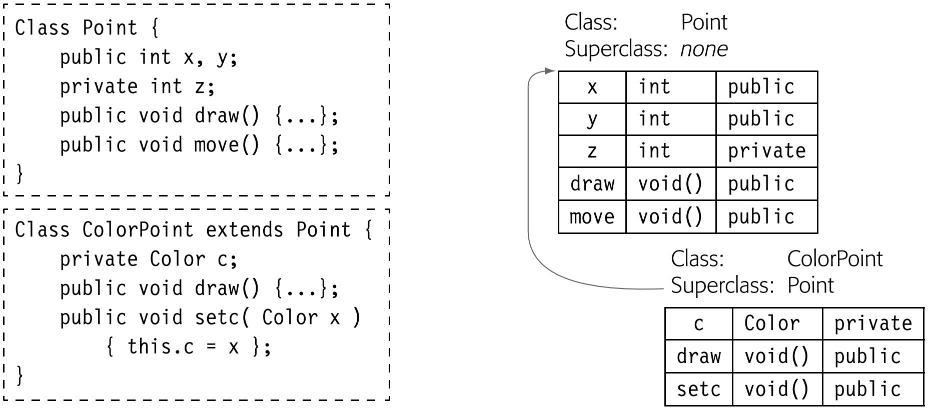
Qualified name
a multipart name, such as `x.part`, where `part` is an element of an aggregate entity named `x`

With an open class structure, the compiler may need to generate code that causes some of this name resolution to occur at runtime, as occurs with a virtual function in C++. In general, runtime name resolution replaces a simple, often inexpensive, reference with a call to a more expensive runtime support routine that resolves the name and provides the appropriate access (read, write, or execute).

One of the primary sources of opportunity for just-in-time compilers is lowering the costs associated with runtime name resolution.

Building the Model

As the parser processes a class definition, it can (1) enter the class name into the current lexical scope and (2) create a new table for the names defined in the class. Since both the contents of the class and its inheritance context are specified with syntax, the compiler writer can use syntax-driven



```
Class Point {
    public int x, y;
    private int z;
    public void draw() {...};
    public void move() {...};
}
```

```
Class ColorPoint extends Point {
    private Color c;
    public void draw() {...};
    public void setc( Color x )
        { this.c = x };
}
```

(a) Class Definitions

Class: Point
Superclass: *none*

x	int	public
y	int	public
z	int	private
draw	void()	public
move	void()	public

Class: ColorPoint
Superclass: Point

c	Color	private
draw	void()	public
setc	void()	public

(b) Corresponding Scope Tables

■ **FIGURE 5.11** Tables for the Inheritance Hierarchy.

actions to build and populate the table and to link it into the surrounding inheritance hierarchy. Member names are found in the inheritance hierarchy; unqualified names are found in the lexical hierarchy.

The compiler can use the symbol-table building blocks designed for lexical hierarchies to represent inheritance hierarchies. Fig. 5.11 shows two class definitions, one for `Point` and another for `ColorPoint`, which is a subclass of `Point`. The compiler can link these tables into a search path for the inheritance hierarchy, shown in the figure as a `SuperClass` pointer. More complicated situations, such as lexically nested class definitions, simply produce more complex search paths.

Examples

Object-oriented languages differ in the vocabulary that they use and in the object-models that they use.

C++ has a closed class structure. By design, method names can be bound to implementations at compile time. C++ includes an explicit declaration to force runtime binding—the C++ virtual function.

By contrast, JAVA has an open class structure, although the cost of changing the class structure is high—the code must invoke the *class loader* to import new class definitions. A compiler could, in principle, resolve method names to implementations at startup and rebind after each invocation of the class loader. In practice, most JAVA systems interpret bytecode and compile frequently executed methods with a just-in-time compiler. This approach allows high-quality code and late binding. If the class loader overwrites some class definition that was used in an earlier JIT-compilation, it can force recompilation by invalidating the code for affected methods.

Multiple Inheritance

Some OOLs allow multiple inheritance. The language needs syntax that lets a programmer specify that members *a*, *b*, and *c* inherit their definitions from superclass *x* while members *d* and *e* inherit their definitions from superclass *y*. The language must resolve or prohibit nonsensical situations, such as a class that inherits multiple definitions of the same name.

Multiple inheritance
a feature that allows a class to inherit from multiple immediate superclasses

To support multiple inheritance, the compiler needs a more complex model of the inheritance hierarchy. It can, however, build an appropriate model from the same building blocks: symbol tables and explicit search paths. The complexity largely manifests itself in the search paths.

5.4.3 Visibility

Programming languages often provide explicit control over visibility—that is, where in the code a name can be defined or used. For example, C provides limited visibility control with the `static` keyword. Visibility control arises in both lexical and inheritance hierarchies.

Visibility
A name is *visible* at point *p* if it can be referenced at *p*.

Some languages provide ways to control a name's visibility.

C's `static` keyword specifies both lifetime and visibility. A C `static` variable has a lifetime of the entire execution and its visibility is restricted to the current scope and any scopes nested inside the current scope. With a declaration outside of any procedure, `static` limits visibility to code within that file. (Without `static`, such a name would be visible throughout the program.)

For a C `static` variable declared inside a procedure, the lifetime attribute of `static` ensures that its value is preserved across invocations. The visibility attribute of `static` has no effect, since the variable's visibility was already limited to the declaring procedure and any scopes nested inside it.

JAVA provides explicit control over visibility via the keywords `public`, `private`, `protected`, and `default`.

public A `public` method or data member is visible from anywhere in the program.

private A `private` method or data member is only visible within the class that encloses it.

protected A `protected` method or data member is visible within the class that encloses it, in any other class declared in the same package, and in any subclass declared in a different package.

default A `default` method or data member is visible within the class that encloses it and in any other class declared in the same package. If no visibility is specified, the object has `default` visibility.

Neither `private` nor `protected` can be used on a declaration at the top level of the hierarchy because they define visibility with respect to the enclosing class; at the top level, a declaration has no enclosing class.

As the compiler builds the naming environment, it must encode the visibility attributes into the name-space model. A typical implementation will include a visibility tag in the symbol table record of each name. Those tags are consulted in symbol table lookups.

As mentioned before, PYTHON determines a variable's visibility based on whether its defining occurrence is a definition or a use. (A use implies that the name is global.) For objects, PYTHON provides no mechanism to control visibility of their data and code members. All attributes (data members) and methods have global visibility.

5.4.4 **Performing Compile-Time Name Resolution**

During translation, the compiler often maps a name's lexeme to a specific entity, such as a variable, object, or procedure. To resolve a name's identity, the compiler uses the symbol tables that it has built to represent the lexical and inheritance hierarchies. Language rules specify a search path through these tables. The compiler starts at the innermost level of the search path. It performs a lookup on each table in the path until it either finds the name or fails in the outermost table.

The specifics of the path are language dependent. If the syntax of the name indicates that it is an object-relative reference, then the compiler can start with the table for the object's class and work its way up the inheritance hierarchy. If the syntax of the name indicates that it is an "ordinary" program variable, then the compiler can start with the table for the scope in which the reference appears and work its way up the lexical hierarchy. If the language's syntax fails to distinguish between data members of objects and ordinary variables, then the compiler must build some hybrid search path that combines tables in a way that models the language-specified scope rules.

The compiler can maintain the necessary search paths with syntax-driven actions that execute as the parser enters and leaves scopes, and as it enters and leaves declarations of classes, structures, and other aggregates. The details, of course, will depend heavily on the specific rules in the source language being compiled.

SECTION REVIEW

Programming languages provide mechanisms to control the lifetime and visibility of a name. Declarations allow explicit specification of a name's properties. The placement of a declaration in the code has a direct effect on lifetime and visibility, as defined by the language's scope rules. In an object-oriented language, the inheritance environment also affects the properties of a named entity.

To model these complex naming environments, compilers use two fundamental tools: symbol tables and search paths that link tables together in a hierarchical fashion. The compiler can use these tools to construct context-specific search spaces that model the source-language rules.

REVIEW QUESTIONS

1. Assume that the compiler builds a distinct symbol table and search path for each scope. For a simple PASCAL-like language, what actions should the parser take on entry to and exit from each scope?
2. Using the table and search path model for name resolution, what is the asymptotic cost of (a) resolving a local name? (b) resolving a nonlocal name? (Assume that table lookup has a $O(1)$ cost.) In programs that you have written, how deeply have you nested scopes?

5.5 TYPE INFORMATION

In order to translate references into access methods, the compiler must know what the name represents. A source language name `fee` might be a small integer; it might be a function of two character strings that returns a floating-point number; it might be an object of class `fum`. Before the front end can emit code to manipulate `fee`, it must know `fee`'s fundamental properties, summarized as its *type*.

Type
an abstract category that specifies properties held in common by all members of the type

Common types include *integer*, *character*, *list*, and *function*.

A type is just a collection of properties; all members of the type have the same properties. For example, an integer might be defined as any whole number i in the range $-2^{63} \leq i < 2^{63} - 1$, or `red` might be a value in the enumerated type `colors` defined as the set { `red`, `yellow`, `blue` }.

Types can be specified by rules; for example, the declaration of a structure in C defines a type. The structure's type specifies the set of declared fields and their order inside the structure; each field has its own type that specifies its interpretation. Programming languages predefine some types, called *base*

We represent the type of a structure as the product of the types of its constituent fields, in order.

types. Most languages allow the programmer to construct new types. The set of types in a given language, along with the rules that use types to specify program behavior, are collectively called a *type system*.

The type system allows both language designers and programmers to specify program behavior at a more precise level than is possible with a context-free grammar. The type system creates a second vocabulary for describing the behavior of valid programs. Consider, for example, the JAVA expression a + b. The meaning of + depends on the types of a and b. If a and b are strings, the + operator specifies concatenation. If a and b are numbers, the + operator specifies addition, perhaps with implicit conversion to a common type. This kind of overloading requires accurate type information.

5.5.1 Uses for Types in Translation

Types play a critical role in translation because they help the compiler understand the meaning and, thus, the implementation of the source code. This knowledge, which is deeper than syntax, allows the compiler to detect errors that might otherwise arise at runtime. In many cases, it also lets the compiler generate more efficient code than would be possible without the type information.

Conformable
We will say that an operator and its operands are conformable if the result of applying the operator to those arguments is well defined.

The compiler can use type information to ensure that operators and operands are conformable—that is, that the operator is well defined over the operands' types (e.g., string concatenation might not be defined over real numbers). In some cases, the language may require the compiler to insert code to convert nonconformable arguments to conformable types—a process called implicit conversion. In other cases (e.g., using a floating-point number as a pointer), the language definition may disallow such conversion; the compiler should, at a minimum, emit an informative error message to give the programmer insight into the problem.

If x is real but provably 2, there are less expensive ways to compute a^x than with a Taylor series.

Type information can lead the compiler to translations that execute efficiently. For example, in the expression a^x, the types of a and x determine how best to evaluate the expression. If x is a nonnegative integer, the compiler can generate a series of multiplications to evaluate a^x. If, instead, x is a real number or a negative number, the compiler may need to generate code that uses a more complex evaluation scheme, such as a Taylor-series expansion. (The more complicated form might be implemented via a call to a support library.) Similarly, languages that allow whole structure or whole array assignment rely on conformability checking to let the compiler implement these constructs in an efficient way.

```
if (tag(a) = tag(b)) then  // take the short path
    switch (tag(a)) into {
        case SHORT:    // use SHORT add
            value(c) ← value(a) + value(b)
            tag(c) ← SHORT
            break
        case INTEGER:  // use INTEGER add
            value(c) ← value(a) + value(b)
            tag(c) ← INTEGER
            break
        case LONG INTEGER:  // use LONG INTEGER add
            value(c) ← value(a) + value(b)
            tag(c) ← LONG INTEGER
            break
    }
else    // take the long path
    (c, tag(c)) ← AddMixedTypes(a, tag(a), b, tag(b))
```

■ **FIGURE 5.12** Integer Addition with Runtime Type Checking.

At a larger scale, type information plays an important enabling role in modular programming and separate compilation. Modular programming creates the opportunity for a programmer to mis-specify the number and types of arguments to a function that is implemented in another file or module. If the language requires that the programmer provide a *type signature* for any externally defined function (essentially, a C function prototype), then the compiler can check the actual arguments against the type signature.

Type signature
a specification of the types of the formal parameters and return value(s) of a function

Function prototype
The C language includes a provision that lets the programmer declare functions that are not present. The programmer includes a skeleton declaration, called a *function prototype*.

Type information also plays a key role in garbage collection (see Section 6.6.2). It allows the runtime collector to understand the size of each entity on the heap and to understand which fields in the object are pointers to other, possibly heap-allocated, entities. Without type information, collected at compile time and preserved for the collector, the collector would need to conservatively assume that any field might be a pointer and apply runtime range and alignment tests to exclude out-of-bounds values.

Lack of Type Information

If type information is not available during translation, the compiler may need to emit code that performs type checking and code selection at runtime. Each entity of unknown type would need a runtime tag to hold its type. Instead of emitting a simple operator, the compiler would need to generate case logic based on the operand types, both to perform tag generation and to manipulate the values and tags.

Complete type information might be unavailable due to language design or due to late binding.

Fig. 5.12 uses pseudocode to show what the compiler might generate for addition with runtime checking and conversion. It assumes three types, *SHORT*, *INTEGER*, and *LONG INTEGER*. If the operands have the same type, the code selects the appropriate version of the addition operator, performs the arithmetic, and sets the tag. If the operands have distinct types, it invokes a library routine that performs the complete case analysis, converts operands appropriately, adds the converted operands, and returns the result and its tag.

By contrast, of course, if the compiler had complete and accurate type information, it could generate code to perform both the operation and any necessary conversions directly. In that situation, runtime tags and the associated tag-checking would be unnecessary.

5.5.2 Components of a Type System

A type system has four major components: a set of base types, or built-in types; rules to build new types from existing types; a method to determine if two types are equivalent; and rules to infer the type of a source-language expression.

Base Types

The size of a "word" may vary across implementations and processors.

Most languages include base types for some, if not all, of the following kinds of data: numbers, characters, and booleans. Most processors provide direct support for these kinds of data, as well. Numbers typically come in several formats, such as integer and floating point, and multiple sizes, such as byte, word, double word, and quadruple word.

Individual languages add other base types. LISP includes both a rational number type and a recursive-list type. Rational numbers are, essentially, pairs of integers interpreted as a ratio. A list is either the designated value `nil` or a list built with the constructor `cons`; the expression `(cons first rest)` is an ordered list where `first` is an object and `rest` is a list.

Languages differ in their base types and the operators defined over those base types. For example, C and C++ have many varieties of integers; `long int` and `unsigned long int` have the same length, but support different ranges of integers. PYTHON has multiple string classes that provide a broad set of operations; by contrast, C has no string type so programmers use arrays of characters instead. C provides a pointer type to hold an arbitrary memory address; JAVA provides a more restrictive model of reference types.

Compound and Constructed Types

The base types of a programming language provide an abstraction for the actual kinds of data supported by the processor. However, the base types are often inadequate to represent the information domain that the programmer needs—abstractions such as graphs, trees, tables, records, objects, classes, lists, stacks, and maps. These higher-level abstractions can be implemented as collections of multiple entities, each with its own type.

Some languages provide higher-level abstractions as base types, such as PYTHON maps.

The ability to construct new types to represent compound or aggregate objects is an essential feature of many languages. Typical constructed types include arrays, strings, enumerated types, and structures or records. Compound types let the programmer organize information in novel, program-specific ways. Constructed types allow the language to express higher-level operations, such as whole-structure assignment. They also improve the compiler's ability to detect ill-formed programs.

In an OOL, classes can be treated as constructed types. Inheritance defines a subtype relationship, or specialization.

Arrays

Arrays are among the most widely used aggregate objects. An array groups together multiple objects of the same type and gives each a distinct name—albeit an implicit, computed name rather than an explicit, programmer-designated name. The C declaration `int a[100][200];` sets aside space for $100 \times 200 = 20{,}000$ integers and ensures that they can be addressed using the name `a`. The references `a[1][17]` and `a[2][30]` access distinct and independent memory locations. The essential property of an array is that the program can compute names for each of its elements by using numbers (or some other ordered, discrete type) as subscripts.

Support for operations on arrays varies widely. FORTRAN 90, PL/I, and APL all support assignment of whole or partial arrays. These languages support element-by-element application of arithmetic operations to arrays. For conformable arrays `x`, `y`, and `z`, the statement `x = y + z` would overwrite each `x[i,j]` with `y[i,j] + z[i,j]`. APL takes the notion of array operations further than most languages; it includes operators for inner product, outer product, and several kinds of reductions. For example, the sum reduction of `y`, written `x ← +/y`, assigns `x` the scalar sum of the elements of `y`.

Array conformability

Two arrays *a* and *b* are *conformable* with respect to some array operator if the dimensions of *a* and *b* make sense with the operator.

Matrix multiply, for example, imposes different conformability requirements than does matrix addition.

An array can be viewed as a constructed type because it is specified with the type of its elements. Thus, a 10×10 array of integers has type *two-dimensional array of integers*. Some languages include the array's dimensions in its type; thus, a 10×10 array of integers has a different type than a 12×12 array of integers. This approach makes array operations where the operands have incompatible dimensions into type errors; thus, they are detected and reported in a systematic way. Most languages allow arrays of any base type; some languages allow arrays of constructed types, such as structures, as well.

Strings

Support for strings varies across languages. Some languages, such as PYTHON or PL/I, support multiple kinds of strings with similar properties, attributes, and operations. Others, such as FORTRAN or C, simply treat a string as a vector of characters.

A true string type differs from an array type in several important ways. Operations that make sense on strings, such as concatenation, translation, and computing string length, may not have analogs for arrays. The standard comparison operators can be overloaded so that string comparisons work in the natural way: `"a" < "boo"` and `"fee" < "fie"`. Implementing a similar comparison for arrays of characters suggests application of the idea to arrays of numbers or structures, where the analogy may not hold. Similarly, the actual length of a string may differ from its allocated size, while most applications of an array use all the allocated elements.

Enumerated Types

Many languages let the programmer construct a type that contains a specific set of constant values. An *enumerated type* lets the programmer use self-documenting names for small sets of constants. Classic examples include the days of the week and the months of the year. In C syntax, these might be written

```
enum WeekDay { Monday, Tuesday, Wednesday,
               Thursday, Friday, Saturday, Sunday};

enum Month   { January, February, March, April,
               May, June, July, August, September,
               October, November, December};
```

The compiler maps each element of an enumerated type to a distinct value. The elements of an enumerated type are ordered, so comparisons between elements of the same type make sense. In the examples, `Monday < Tuesday` and `June < July`. Operations that compare different enumerated types make no sense—for example, `Tuesday > September` should produce a type error. PASCAL ensures that each enumerated type behaves as if it were a subrange of the integers. For example, the programmer can declare an array indexed by the elements of an enumerated type.

Structures and Variants

Structures, or *records*, group together multiple objects of arbitrary type. The elements of the structure are typically given explicit names. For example, a programmer implementing a parse tree in C might need nodes with both one and two children.

```
struct N1 {              struct N2 {             union Node {
  int   Operator;          int   Operator;         struct N1 one;
  int   Value;             int   Value;            struct N2 two;
  union Node *left;        union Node *left;     };
};                         union Node *right;
                         };
```

The type of a structure is the ordered product of the types of the elements that it contains. We might describe N1 and N2 as:

```
N1:   int × int × (Node *)
N2:   int × int × (Node *) × (Node *)
```

These new types should have the same essential properties that a base type has. In C, autoincrementing a pointer to an N1 or casting a pointer into an (N1 *) has the desired effect—the behavior is analogous to what happens for a base type.

The example creates a new type, Node, that is a structure of either type N1 or type N2. Thus, the pointer in an N1 node can reference either an N1 node or an N2 node. PASCAL creates unions with variant records. C uses a union. The type of a union is the alternation of its component types; thus, Node has type N1 ∪ N2.

Between them, the language and the runtime need a mechanism to disambiguate references. One solution is fully qualified references as in p→Node.N1.Value versus p→Node.N2.Value. Alternatively, the language might adopt PASCAL's strategy and require runtime tags for variant records, with explicit checks for the tags at runtime.

Objects and Classes

In an object-oriented language, classes define both the content and form of objects, and they define the inheritance hierarchy that is used to resolve object-relative references. In implementation, however, an object looks like a record or structure whose organization is specified by the class definition.

AN ALTERNATIVE VIEW OF STRUCTURES

The classical view of structures treats each kind of structure as a distinct type. This approach to structure types follows the treatment of other aggregates, such as arrays and strings. It seems natural. It makes distinctions that are useful to the programmer. For example, a tree node with two children probably should have a different type than a tree node with three children; presumably, they are used in different situations. A program that assigns a three-child node to a two-child node should generate a type error and a warning message to the programmer.

From the runtime system's perspective, however, treating each structure as a distinct type complicates matters. With distinct structure types, the heap contains a set of objects drawn from an arbitrary set of types. This makes it difficult to reason about programs that deal directly with the objects on the heap, such as a garbage collector. To simplify such programs, their authors sometimes take a different approach to structure types.

This alternate model considers all structures in the program as a single type. Individual structure declarations each create a variant form of the type *structure*. The type *structure*, itself, is the union of all these variants. This approach lets the program view the heap as a collection of objects of a single type, rather than a collection of many types. This view makes code that manipulates the heap simpler to analyze and optimize.

Type Equivalence

```
struct Tree {
  int value;
  struct Tree *left;
  struct Tree *right;
}

struct Bush {
  int value;
  struct Bush *left;
  struct Bush *right;
}
```

The compiler needs a mechanism to determine if two constructed types are equivalent. (The answer is obvious for base types.) Consider the C structure declarations shown in the margin. Are Tree and Bush the same type? Are they equivalent? Any language that includes constructed types needs an unambiguous rule to answer this question. Historically, languages have taken one of two approaches.

1. *Name Equivalence* asserts that two types are equivalent if and only if the programmer calls them by the same name. This approach assumes that naming is an intentional act and that the programmer uses names to impart meaning.
2. *Structural Equivalence* asserts that two types are equivalent if and only if they have the same structure. This approach assumes that structure matters and that names may not.

Tree and Bush have structural equivalence but not name equivalence.

Each approach has its adherents and its detractors. However, the choice between them is made by the language designer, not the compiler writer. The

Production			Syntax-Driven Action
Expr	→	*Expr* + *Term*	{ set_type(\$\$,$\mathcal{F}_{+}$(type(\$1),type(\$3))); };
	|	*Expr* − *Term*	{ set_type(\$\$,$\mathcal{F}_{-}$(type(\$1),type(\$3))); };
	|	*Term*	{ set_type(\$\$,type(\$1)); };
Term	→	*Term* × *Factor*	{ set_type(\$\$,$\mathcal{F}_{\times}$(type(\$1),type(\$3))); };
	|	*Term* ÷ *Factor*	{ set_type(\$\$,$\mathcal{F}_{\div}$(type(\$1),type(\$3))); };
	|	*Factor*	{ set_type(\$\$,type(\$1)); };
Factor	→	(*Expr*)	{ set_type(\$\$,type(\$2)); };
	|	num	{ set_type(\$\$,type(num)); };
	|	name	{ set_type(\$\$,type(name)); };

■ **FIGURE 5.13** Framework to Assign Types to Subexpressions.

compiler writer must implement an appropriate representation for the type and an appropriate equivalence test.

5.5.3 **Type Inference for Expressions**

The compiler must assign, to each expression and subexpression, a specific type. The simplest expressions, names and nums, have well defined types. For expressions computed from references, the compiler must infer the type from the combination of the operation and the types of its operands.

The relationship between operator, operand types, and result type must be specified for the compiler to infer expression types. Conceptually, we can think of the relationship as a recursive function over the tree; in practice, the rules vary from simple and obvious to arcane. The digression on page 248 describes the rules for expressions in C++. Because C++ has so many base types, its rules are voluminous.

The result type of an expression depends on the operator and the types of its operands. The compiler could assign types in a bottom-up walk over an expression tree. At each node, it would set the node's type from the type of its operator and its children. Alternatively, the compiler could assign types as part of its syntax-driven framework for translation.

Fig. 5.13 sketches the actions necessary to assign types to subexpressions in a syntax-driven framework. It assumes that the type-function for an operator α is given by a function $\mathcal{F}_{\alpha}$. Thus, the type of a multiplication is just $\mathcal{F}_{\times}(t_1, t_2)$, where t_1 and t_2 are the types of the left and right operands of $\times$. Of course, the compiler writer would likely pass a structured value on the stack, so the references to \$\$, \$1, and \$3 would be more complex.

NUMERICAL CONVERSIONS IN C++

C++, as defined in the ISO 2017 standard, has a large and complex set of conversion rules [212]. Here is a simplified version of the promotion and conversion rules for numerical values.

Integral Promotion: A character value or a value in an untyped enumeration can be promoted to the first integer type that will hold all of its values. The integer types, in order, are: `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, and `unsigned long long int`. (For a typed enumeration, conversion is legal only if the underlying type converts to integer.)

Floating-Point Promotion: A `float` value can be promoted to type `double`.

Integer Conversions: A value of an integer type can be converted to another integer type, as can a value of an enumeration type. For an unsigned destination type, the result is the smallest unsigned integer congruent to the source value. For a signed destination type, the value is unchanged if it fits in the destination type, otherwise the result is *implementation-defined*.

Floating-Point Conversions: A value of floating-point type can be converted to another floating-point type. If the destination type can exactly represent the source value, the result is that value. Otherwise it is an implementation-defined choice between the two adjacent values.

Boolean Conversion: A numerical value, enumeration value, or pointer value can be converted to a value of type `bool`. A value of zero, a null pointer, or a null member pointer all convert to `false`; any other value converts to `true`.

The compiler tries to convert the source value to the destination type, which may involve both a promotion and a conversion.

In a language with more complex inference rules, the compiler might build an IR that has incomplete type information and perform one or more passes over the IR to assign types to subexpressions.

The Role of Declarations

Programming languages differ on whether or not they require declarations. In a language with mandatory declarations, the declarations establish a concrete type for every named entity; those types serve, in turn, as the initial information for type inference. In a language without declarations, such as PYTHON or LISP, the compiler must infer types for values from the context in which they appear in the code. For example, the assignment `fee` ← `'a'` might imply that fee has a type that can hold a single character, while fee ← "a" implies that fee can hold a character string.

This example assumes the C convention of single quotes for characters and double quotes for strings.

Programming languages also differ on where in the code a declaration must appear. Many languages have a "declare before use" rule; any name must be declared before it appears in the executable code. This rule facilitates type-checking during the parser's translation into an initial IR form. Languages that do not require declaration before use force the compiler to build an initial IR that abstracts away details of type, and to subsequently perform type inference and checking on that abstract IR so that the compiler can refine operators and references to reflect the correct type information.

Mixed-Type Expressions

Programming languages differ on the extent to which they expect the compiler to insert type conversions when the code specifies an expression with types that are not directly compatible. For example, an expression a × b may be defined for the case when a and b are both integers or both floating-point numbers, but not when a is an integer and b is a floating-point number. The language may require the compiler to report an error; alternatively, it might require the compiler to insert a conversion. Section 7.2.2 discusses the implementation of implicit conversions.

For example, ANSI C++ supports multiple kinds of integers that differ in the range of numbers that each can represent. The language definition requires that the compiler insert code to convert between these representations; the definition specifies the behavior with a set of rules. Its rules specify the conversions for the division of an integer by a floating-point number and forbid division by a character string.

Interprocedural Aspects of Type Inference

Type inference for expressions depends, inherently, on the other procedures that form the executable program. In even the simplest type systems, expressions contain function calls. The compiler must check each of those calls. It must ensure the type compatibility of each actual parameter with the corresponding formal parameter. It must determine the type of the returned value for use in further inference.

To analyze and understand procedure calls, the compiler needs a *type signature* for each function. For example, in C's standard library, strlen computes a character string's length. Its function prototype is:

```
    unsigned int strlen(const char *s);
```

This prototype asserts that `strlen` takes an argument of type `char *`. The `const` attribute indicates that `strlen` does not modify `s`. It returns a nonnegative integer. The type signature might be written:

*strlen : const char * → unsigned int*

which we read as "`strlen` is a function that takes a constant-valued character string and returns an unsigned integer."

As a second example, `filter` in SCHEME has the type signature:

$$filter: (\alpha \rightarrow boolean) \times list\ of\ \alpha \ \rightarrow\ list\ of\ \alpha$$

`filter` returns a list that contains every element of the input list for which the input function returns `true`.

That is, `filter` is a function of two arguments. The first should be a function that maps some type α into a boolean, written $(\alpha \rightarrow boolean)$, and the second should be a list whose elements are of the same type α. Given arguments of those types, `filter` returns a list whose elements have type α. The function `filter` exhibits *parametric polymorphism*: its result type is a function of its argument types.

To perform accurate type inference, the compiler needs a type signature for every function. It can obtain that information in several ways. The compiler can require that the entire program be present for compilation, eliminating separate compilation. The compiler can require a type signature for each function, typically done with mandatory function prototypes. The compiler can defer type checking until link time or runtime, when such information is available. Finally, the compiler writer can embed the compiler in a program-development system that gathers the requisite information. Each of these approaches has been used in real systems.

SECTION REVIEW

A type represents a set of properties common to all values of that type. A type system assigns a type to each value in a program. Programming languages use types to define legal and illegal behavior. A good type system can increase language expressiveness, expose subtle errors, and let the compiler avoid runtime type checks.

A type system consists of a set of base types, rules to construct new types from existing ones, a method to determine the equivalence of two types, and rules to infer the type of an expression. The notions of base types, constructed types, and type equivalence should be familiar from most high-level languages.

REVIEW QUESTIONS

1. For your favorite programming language, what are its base types? Is there a mechanism to build an aggregate type? Does it provide a mechanism for creating a procedure that takes a variable number of arguments, such as `printf` in the C standard I/O library?
2. Type safety at procedure calls is often based on the use of prototypes—a declaration of the procedure's arguments and return values. Sketch a mechanism that could ensure the validity of those function prototypes.

Hint: It may require interaction with the linker or the runtime system.

5.6 STORAGE LAYOUT

Given a model of the name space and type information for each named entity, the compiler can perform storage layout. The process has two steps. First, the compiler must assign each entity to a logical data area. This decision depends on both the entity's lifetime and its visibility. Second, for each logical data area, the compiler assigns each entity in that area an offset from the data area's start.

5.6.1 Storage Classes and Data Areas

The compiler can classify values that need storage by their lifetimes. Most programming languages let programmers create values in at least the following storage classes: *automatic*, *static*, and *irregular*. The compiler maps a specific variable name into a storage area based on its lifetime, storage class, and visibility (see Section 4.7.3).

Automatic Variables

An automatic variable *a* has a lifetime that is identical to the lifetime of its declaring scope. Therefore, it can be stored in the scope's local data area. For example, if *a* is declared in procedure *p*, the compiler can store *a* in *p*'s local data area. (If the scope is contained in *p*, the compiler can set aside space for the scope inside *p*'s local data area.) If *a* is local, scalar, and unambiguous, the compiler may choose to store it in a register (see Section 4.7.2).

To manage the execution of procedure *p*, the compiler must ensure that each invocation of *p* has a small block of storage to hold the control information needed by the call and return process. This *activation record* (AR) will also contain the arguments passed to *p* as parameters. ARs are, in principle and in practice, created when control enters a procedure and freed when control exits that procedure.

Activation record
a region of memory set aside to hold control information and the local data area for an invocation of a procedure

We treat "activation" and "invocation" as synonyms.

Activation record pointer
At runtime, the code will maintain a pointer to the current AR. The activation record pointer (ARP) almost always resides in a register for quick access.

The compiler can place *p*'s local data area inside its AR. Each call to *p* will create a new AR and, with it, a new local data area. This arrangement ensures that the local data area's lifetime matches the invocation's lifetime. It handles recursive calls correctly; it creates a new local data area for each call. Placing the local data area in the AR provides efficient access to local variables through the activation record pointer (ARP). In most implementations, the local data area occupies one end of the procedure's AR (see Section 6.3.1).

Static Variables

A static variable *s* has a lifetime that runs from the first time the executing program defines *s* through the last time that the execution uses *s*'s value. The first definition and last use of *s* could cover a short period in the execution; they could also span the entire execution. The attribute `static` is typically implemented to run from the start of execution to its end.

Programming languages support static variables with a variety of visibility constraints. A global variable is static; it has visibility that spans multiple, nonnested procedures. A static variable declared inside a procedure has procedure-wide visibility (including nested scopes); the variable retains its value across multiple invocations of the procedure, much like a global variable. C uses `static` to create a file-level visibility; the value is live for the entire execution but only visible to procedures defined inside the same file.

Compilers create distinct data areas for static variables. In principle, a program could implement individual data areas for each static variable; alternatively, it could lump them all together into a single area. The compiler writer must develop a rationale that determines how static variables map into individual data areas. A simple approach is to create a single static data area per file of code and rely on the compiler's name resolution mechanism to enforce visibility constraints.

Compilers typically use assembly language constructs to create and initialize static data areas, so allocation, initialization, and deallocation have, essentially, no runtime cost. The compiler must create global data areas in a way that allows the system's linker to map all references to a given global name to the same storage location—the meaning of "global" visibility.

Irregular Entities

If a heap-allocated value has exactly one allocation, either the programmer or the compiler can convert it to a static lifetime.

Some values have lifetimes that are under program control, in the sense that the code explicitly allocates space for them. (Deallocation may be implicit or explicit.) The key distinction is that allocation and deallocation occur at times unrelated to any particular procedure's lifetime and have the potential to occur multiple times in a single execution.

The compiler's runtime support library must provide a mechanism to allocate and free these irregular entities. Systems commonly use a *runtime heap* for such purposes. Control of the heap may be explicit, through calls such as LINUX's `malloc` and `free`. Alternatively, the heap may be *managed* with implicit deallocation through techniques such as garbage collection or reference counting.

Heap
a region of memory set aside for irregular entities and managed by the runtime support library

While storage for the actual entities may be on the heap, the source code typically requires a name to begin a reference or chain of references. Thus, a linked list might consist of an automatic local variable, such as `root`, that contains a pointer to the first element of the list. `root` would need space in a register or the local data area, while the individual list elements might be allocated on the heap.

Temporary Values

During execution, a program computes many values that are never stored into named locations. For example, when compiled code evaluates `a - b × c`, it computes the value of `b × c` but has no semantic reason to retain its value. Because these temporary values have no names, they cannot be reused by the programmer. They have brief lifetimes.

Optimization can extend a temporary value's lifetime. If the code recomputes `b × c`, the compiler might preserve its value rather than compute it twice (see Section 8.4.1).

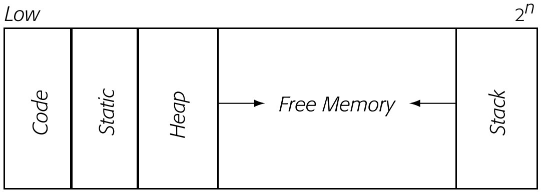
When a temporary value has a representation that can fit in a register, the compiler should try to keep that value in a register. Some temporary values cannot fit in a register. Others have unknown lengths. For example, if `d` and `e` are strings of unknown length and `+` is concatenation, then one scheme to evaluate `length(d + e)` creates the string temporary `d + e`, also of unknown length.

The compiler can place large values of known or bounded length at the end of the local data area. If the length cannot be bounded, the compiler may need to generate code that performs a runtime allocation to create space for the value on the heap.

5.6.2 Layout Within a Virtual Address Space

The compiler must plan how the code will use memory at runtime. In most systems, each program runs in a distinct *virtual address space*; the program executes in its own protected range of addresses. The operating system and the underlying hardware map that virtual address space onto the actual physical hardware in a transparent fashion; the compiler only concerns itself with virtual addresses.

Virtual address space
In many systems, each process has an address space that is isolated from those of other processes. These address spaces are *virtual*, in the sense that they are not tied directly to physical memory (see Fig. 5.15).



■ **FIGURE 5.14** Virtual Address-Space Layout.

The layout of the virtual address space is determined by an agreement among the operating system, hardware, and compiler. While minor details differ across implementations, most systems resemble the layout shown Fig. 5.14. The address space divides into four categories of storage:

Code At one end of the address space, the compiler places executable code. Compiled code has, in general, known size. It rarely changes at runtime. If it changes size at runtime, the new code generally lives in a heap-allocated block of storage.

Static The second category of storage holds statically defined entities. This category includes global and static variables. The size of the static area can be determined at *link time*, when all of the code and data is combined to form an executable image.

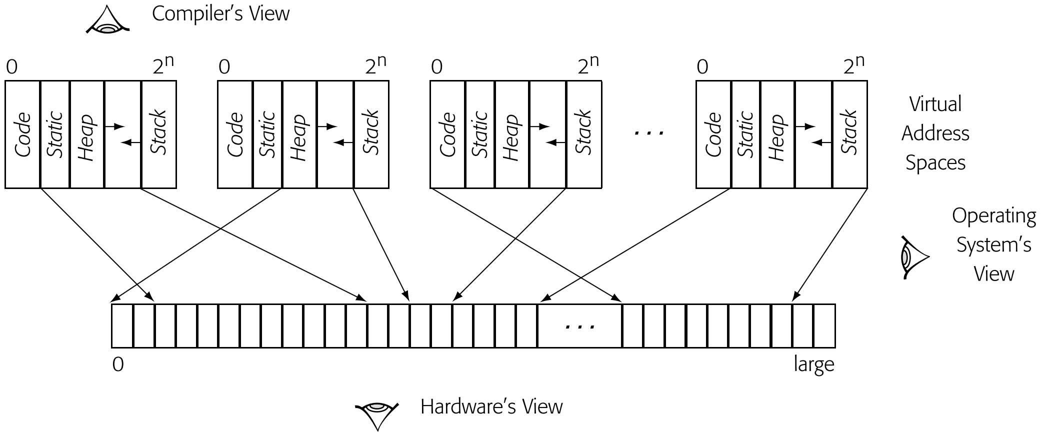
In some circumstances, activation records must be heap allocated (see Section 6.3.1).

Heap The heap is a variable-sized region of memory allocated under explicit program control. Dynamically allocated entities, such as variable-sized data structures or objects (in an OOL), are typically placed in the heap. Deallocation can be implicit, with garbage collection or reference counting, or explicit, with a runtime support routine that frees a heap-allocated object.

Stack Most of the time, procedure invocations obey a *last-in, first-out* discipline. That is, the code calls a procedure and that procedure returns. In this environment, activation records can be allocated on a stack, which allows easy allocation, deallocation, and reuse of memory. The stack is placed opposite the heap, with all remaining free space between them.

The heap and the stack grow toward each other. This arrangement allows for efficient use of the free space between them.

From the compiler's perspective, this virtual address space is the whole picture. However, modern computer systems typically execute many programs in an interleaved fashion. The operating system maps multiple virtual address spaces into the single physical address space supported by the processor. Fig. 5.15 shows this larger picture. Each program is isolated in its own virtual address space; each can behave as if it has its own machine.



■ **FIGURE 5.15** Different Views of the Address Space.

A single virtual address space can occupy disjoint pages in the physical address space; thus, the addresses 100,000 and 200,000 in the program's virtual address space need not be 100,000 bytes apart in physical memory. In fact, the physical address associated with the virtual address 100,000 may be larger than the physical address associated with the virtual address 200,000. The mapping from virtual addresses to physical addresses is maintained cooperatively by the hardware and the operating system. It is, in almost all respects, beyond the compiler's purview.

Page
the fundamental unit of allocation in a virtual address space

The operating system maps virtual pages into physical page frames.

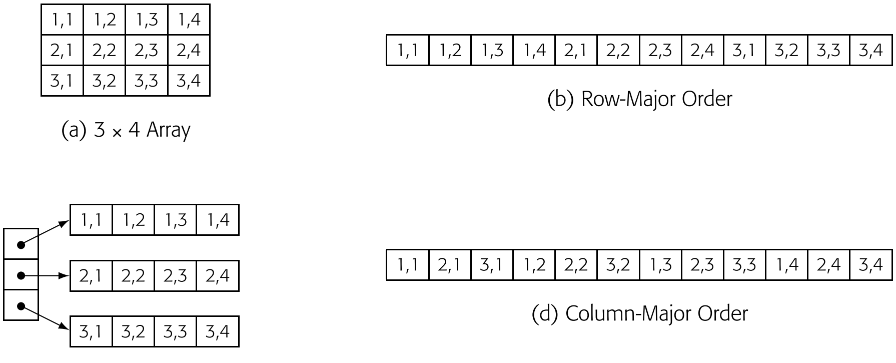
5.6.3 Storage Assignment

Given the set of variables in a specific data area, the compiler must assign them each a storage location. If the compiler writer intends to maximize register use, then the compiler will first find each register-sized unambiguous value and assign it a unique virtual register (see Section 4.7.2). Next, it will assign each ambiguous value an offset from the start of the data area. Section 5.6.5 describes a method for laying out data areas while minimizing the impact of hardware alignment restrictions.

Internal Layout for Arrays

Most programming languages include an array construct—a dimensioned aggregate structure in which all the members have the same type. During storage layout, the compiler needs to know where it will place each array. It must also understand when the size of that array is set and how to calculate its space requirements. These issues depend, in part, on the scheme used to lay out the array elements in memory.

While arrays were added to FORTRAN to model matrices in numerical calculations, they have many other uses.



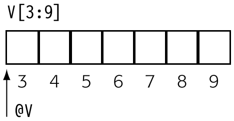
(a) 3 × 4 Array

(b) Row-Major Order

(c) Indirection Vectors

(d) Column-Major Order

■ **FIGURE 5.16** Two-Dimensional Array Layouts.



Vector Layout

The compiler can lay out a one-dimensional array, or *vector*, as a set of adjacent memory locations. Given the range of valid indices, from *low* to *high*, the vector will need $(high - low + 1) \times w$ contiguous bytes of storage, where w is the width of an element in bytes. The address of `V[i]` is just `@V` + $(i - low) \times w$ where `@V` is the address of the first element of `V`.

Section 7.3.2 discusses the address calculations for each of these layouts.

With two or more dimensions, the language must specify an array layout. Fig. 5.16 shows three options that are used in practice. Panel (a) shows a conceptual view of a 3 × 4 array.

An array in *row-major order* is laid out as a series of rows, as shown in panel (b). Many languages use row-major order. Alternatively, an array that is in *column-major order* is laid out as a series of columns, as shown in panel (d). FORTRAN uses column-major order. If the array has c columns and r rows with elements of w bytes, both of these layouts use $c \times r \times w$ bytes of contiguous storage.

The final option is to lay out the array as a series of indirection vectors, as shown in panel (c). JAVA uses this scheme. Here, the final dimension of the array is laid out in contiguous locations, and the other dimensions are represented with vectors of pointers. For an array with c columns and r rows, it requires $c \times r \times w$ space for the data, plus $r \times p$ space for the pointers, where w is the size of an array element and p is the size of a pointer. The individual rows and the column of pointers need not be contiguous.

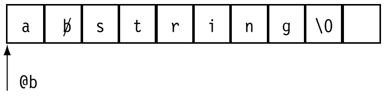
Internal Layout for Strings

Section 7.6 discusses operations on strings.

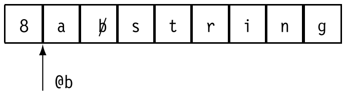
Most programming languages support some form of string. Character strings are common; strings with elements of other types do occur. The representation is slightly more complex than that of a vector because a string variable might take on string values of different lengths at runtime. Thus, a string representation must hold the string's current content and the length of that content. It might also indicate the longest string that it can hold.

Two common representations are a null-terminated string and a string with a length field. A null-terminated string, shown to the left, uses a vector of elements, with a designated end-of-string marker. C introduced this representation; other languages have followed.

The glyph b̸ represents a blank.



String with Null Termination



String with Explicit Length Field

The explicit length representation, shown on the right, stores the value of the length in a separate field. These two layouts have slightly different space requirements; the null-terminated string requires an extra element to mark the string's end while the explicit length representation needs an integer large enough to hold the maximum string length.

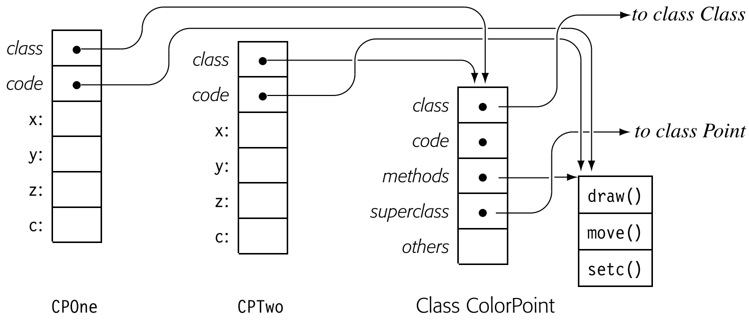
The real difference between these representations lies in the cost of computing the string's length. In the null-terminated string, the cost is $\mathrm{O}(n)$ where n is the string's length, while the same operation is $\mathrm{O}(1)$ in the explicit-length string. This difference carries into other operations that need to know the length, such as concatenation. It plays a critical role in range checking (see Section 7.3.3).

Internal Layout for Structures

The compiler must also perform layout for structures and objects. Most languages treat the interior of a structure declaration as a new scope. The programmer can use arbitrary names for the fields and scope rules will ensure the correct interpretation. Each field in a structure declaration allocates space within the structure; the compiler must assign each field an offset within the structure's representation.

Programming languages differ as to whether or not the text of a structure declaration also defines the layout of the structure. Strong arguments exist for either choice. If the declaration dictates layout, then the compiler assigns offsets to the fields as declared. If the compiler controls structure layout, it can assign offsets within the structure to eliminate wasted space, using the technique for data-area layout.

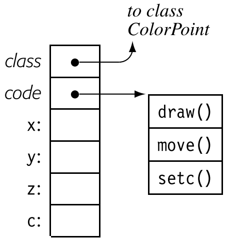
Systems programming languages often follow declaration layout so that a program can interface with hardware defined layouts, such as device control blocks.



■ FIGURE 5.17 Multiple Instances of Class ColorPoint.

Internal Layout for Object Records

In an object-oriented language, each object has its own object record (OR). Because object lifetimes are irregular, ORs typically live on the heap. The OR holds the data members specified by the object's class, along with pointers to its class and, in many implementations, a vector of the class' methods. With inheritance, the OR must include data members inherited from its superclasses and access to code members of its superclasses.



Object Record for Instance Of Class ColorPoint

The drawing in the margin shows an OR layout for an instance of class `ColorPoint` from Fig. 5.11. The OR has storage for each data member of the object, plus pointers to class `ColorPoint`'s OR and to a vector of visible methods for `ColorPoint`.

The major complication in object layout arises from the fact that superclass methods should work on subclass objects. To ensure this interoperability, the subclass object layout must assign consistent offsets to data members from superclasses. With single-inheritance, the strategy of *prefix layout* achieves this goal. The subclass object layout uses the superclass object layout as a prefix. Data members from ancestors in the superclass chain retain consistent offsets; data members from the current class are added to the end of the OR layout.

We use the terms *method vector* and *code vector* interchangeably.

To reduce storage requirements, most implementations store the method vector in the class' OR rather than keeping a copy in each object's OR. Fig. 5.17 shows the ORs for two instances of `ColorPoint` along with the class' OR. Linking the ORs for `CPOne` and `CPTwo` directly to the method vector for `ColorPoint` reduces the space requirement without any direct cost. Of course, offsets in the method vectors must be consistent up the inheritance hierarchy chain; again, prefix layout works well for single inheritance environments.

DETAILS MATTER

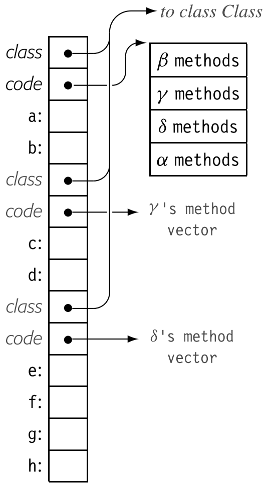
In compiler construction, the details matter. As an example, consider two classes, α and its subclass β. When the compiler lays out β's object records, does it include `private` members of α? Since they are private, an object of class β cannot access them directly.

An object of class β will need those private members from α if α provides public methods that read or write those private members. Similarly, if the OR layout changes without them, the private members may be necessary to ensure that public members have the correct offsets in an OR of class β (even if there is no mechanism to read their values).

To simplify lookup, the OR can contain a fully instantiated code vector, with pointers to both class and superclass methods.

Object Record Layout for Multiple Inheritance

Multiple inheritance complicates OR layout. The compiled code for a superclass method uses offsets based on the OR layout of that superclass. Different immediate superclasses may assign conflicting offsets to their members. To reconcile these competing offsets, the compiler must adopt a slightly more complex scheme: it must use different OR pointers with methods from different superclasses.



Object Record for α

Consider a class α that inherits from multiple superclasses, β, γ, and δ. To lay out the OR for an object of class α, the implementation must first impose an order on α's superclasses—say β, γ, δ. It then lays out the OR for class α with the entire OR for β, including class pointer and method vector, as a prefix to α. Following that, it lays out the OR for γ and, then, the OR for δ. To this layout, it appends the data members of α. It constructs a method vector by appending the inherited methods, in order by class, followed by any methods from α. The drawing in the margin shows this layout, with the class pointers and method vectors for β and γ in the middle of the OR.

The drawing assumes that each class defines two data members: β defines `a` and `b`; γ defines `c` and `d`; δ defines `e` and `f`; and α defines `g` and `h`. The code vector for α points to a vector that contains all of the methods that α defines or inherits.

At runtime, a method from class β will find all of the data members that it expects at the same offsets as in an object of class β. Similarly, a method compiled for class α will find the data members of α at offsets known when the method was compiled.

Methods compiled for β, γ, or δ cannot see members defined in α. Thus, the code can adjust the OR pointer with impunity.

For members of classes γ or δ, however, data members are at the wrong offset. The compiler needs to adjust the OR pointer so that it points to the appropriate point in the OR. Many systems accomplish this effect with a *trampoline function*. The trampoline function simply increments the OR pointer and then invokes the method; on return from the method, it decrements the OR pointer and returns.

5.6.4 Fitting Storage Assignment into Translation

The compiler writer faces a choice in translation. She can design the compiler to perform as much translation as possible during the syntax-driven phase, or she can design it to build an initial IR during the translation and rely on subsequent passes over the IR to complete the translation. The timing of storage layout plays directly into this choice.

The compiler writer can use a mid-production action in a rule similar to

Body → *Decls Execs*

where *Decls* derives declarations and *Execs* derives executable statements.

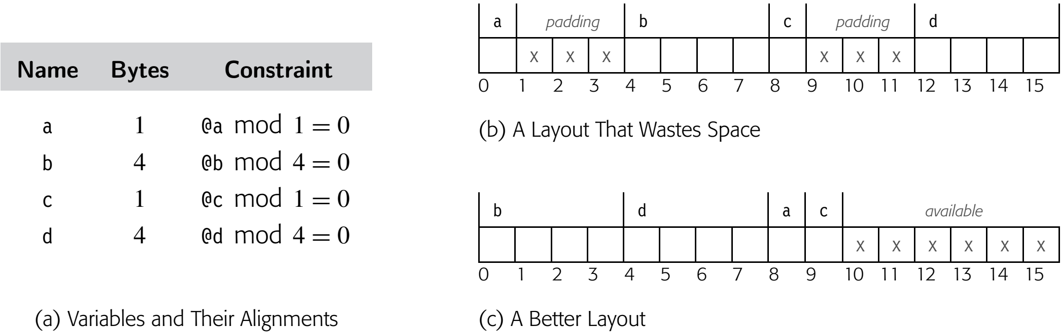
1. Some languages require that all variables be declared before any executable statement appears. The compiler can gather all of the type and symbol information while processing declarations. Before it processes the first executable statement, it can perform storage layout, which allows it to generate concrete code for references.
2. If the language requires declarations, but does not specify an order, the compiler can build up the symbol table during parsing and emit IR with abstract references. After parsing, it can perform type inference followed by storage layout. It can then refine the IR and make the references more concrete.
3. If the language does not require declarations, the compiler must build an IR with abstract references. The compiler can then perform some more complex (probably iterative) type inference on the IR, followed by storage layout. Finally, it can refine the IR and make the references more concrete.

The choice between these approaches depends on the rules of the source language and the compiler writer's preference. A multipass approach may simplify the code in the compiler itself.

5.6.5 Alignment Restrictions and Padding

Alignment restriction
Most processors restrict the alignment of values by their types. For example, an eight-byte integer may need to begin at an address a such that $a \bmod 8 = 0$.

Instruction set architectures restrict the alignment of values. (Assume, for this discussion, that a byte contains eight bits and that a word contains four bytes.) For each hardware-supported data type, the ISA may restrict the set of addresses where a value of that type may be stored. For example, a 32-bit floating-point number might be restricted to begin on a word, or 32-bit, boundary. Similarly, a 64-bit integer might be restricted to a doubleword, or 64-bit boundary.



Name	Bytes	Constraint
a	1	`@a` mod $1 = 0$
b	4	`@b` mod $4 = 0$
c	1	`@c` mod $1 = 0$
d	4	`@d` mod $4 = 0$

(a) Variables and Their Alignments

(b) A Layout That Wastes Space

(c) A Better Layout

■ **FIGURE 5.18** Alignment Issues in Data-Area Offset Assignment.

The compiler has two mechanisms to enforce alignment restrictions. First, it can control the alignment of the start of each data area. Most assembly languages have directives to enforce doubleword or quadword alignment at the start of a data area. Such pseudooperations ensure that each data area starts at a known alignment.

Second, the compiler controls the internal layout of the data area; that is, it assigns an offset to each value stored in the data area. It can ensure, through layout, that each value has the appropriate alignment. For example, a value that needs doubleword alignment must have an offset that is evenly divisible by eight.

Consider a variable x stored at *offset* in a data area that starts at address *base*. If *base* is quadword aligned, then $base \text{ MOD } 16 = 0$. If $offset \text{ MOD } 8 = 0$, then the address of x, which is $base + offset$, is doubleword aligned—that is $(base + offset) \text{ MOD } 8 = 0$.

As the compiler lays out a data area, it must satisfy all of the alignment restrictions. To obtain proper alignment, it may need to insert empty space between values. Fig. 5.18(a) shows the lengths and constraints for a simple four-variable example. Panel (b) shows the layout that results if the compiler assigns them offsets in alphabetical order. It uses sixteen bytes and wastes six bytes in padding. Panel (c) shows an alternative layout that uses ten bytes with no padding. In both cases, some space may be wasted before the next entity in memory.

To create the layout in panel (c), the compiler can build a list of names for a given data area and sort them by their alignment restrictions, from largest to smallest alignment boundary. Next, it can assign offsets to the names in sorted order. If it must insert padding to reach the alignment boundary for the next name, it may be able to fill that space with small-boundary names from the end of the list.

SECTION REVIEW

The compiler must decide, for each runtime entity, where in storage it will live and when its storage will be allocated. The compiler bases its decision on the entity's lifetime and its visibility. It classifies names into storage classes. For objects with predictable lifetimes, the storage class guides these decisions.

The compiler typically places items with unpredictable lifetimes on the runtime heap. Heap-based entities are explicitly allocated; typically, references to heap-based entities involve a level of indirection through a variable with a regular lifetime.

REVIEW QUESTIONS

1. In C, a file might contain both file static and procedure static variables. Does the compiler need to create separate data areas for these two distinct classes of visibility?
2. Consider the short fragment of C code shown in the margin. It names three values, a, b, and *b. Which of these values are ambiguous? Which are unambiguous?

```
void fee() {
    int a, *b;
    ...
    b = &a;
    ...
}
```

5.7 ADVANCED TOPICS

This chapter has focused on the mechanism of syntax-driven translation and its uses in building a compiler. As use cases, it has discussed translation of expressions and if–then–else statements, models of the source program's naming environment, a simple approach to type checking, and storage layout. This section expands on three issues.

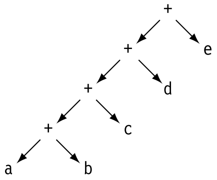
The first subsection looks at the relationship between the direction of recursion in a grammar and associativity. The second subsection discusses the interaction between language design and type inference and checking. The final subsection looks briefly at the interaction between cache offsets and performance.

5.7.1 Grammar Structure and Associativity

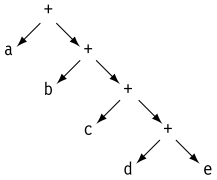
In Chapter 3, we saw left-recursive and right-recursive variants of the expression grammar, along with a transformation to eliminate left-recursion. In that discussion, we noted that the transformation preserves associativity. This subsection explores the relationship between recursion, associativity, IR structure, and parse stack depth. Consider the following simple grammars, for addition over names.

Expr	→	*Expr* `+ name`	*Expr*	→	`name +` *Expr*
	|	`name`		|	`name`

Left-Recursive Grammar   Right-Recursive Grammar



Left-Recursive AST



Right-Recursive AST

Given an input expression `a + b + c + d + e`, the two grammars lead to significantly different ASTs, as shown in the margin. With extreme values, these trees can evaluate to measurably different results.

A postorder evaluation of the AST from the left-recursive grammar will evaluate to `(((a + b) + c) + d) + e`, while the right-recursive version will evaluate to `(((d + e) + c) + b) + a`. With addition, which is both commutative and associative, the numerical difference in these sums will only arise with extremely large or small values.

With an LL(1) parser generator, where left recursion is not an option, the compiler writer can obtain left-associativity by writing the left-recursive grammar and using the transformation to convert left-recursion to right-recursion. With an LR(1) parser generator, the compiler writer can choose either left or right recursion to suit the circumstances.

Stack Depth

In general, left recursion can lead to smaller stack depths. Consider what happens when an LR(1) parser processes the expression `a + b + c + d + e` with each of our grammars shown earlier.

1. *Left-Recursive Grammar* This grammar shifts `a` onto the stack and immediately reduces it to *Expr*. Next, it shifts `+` and `b` onto the stack and reduces *Expr* `+ b` to *Expr*. It continues, shifting a `+` and a `name` onto the stack and reducing the left context to *Expr*. When it hits the end of the string, the maximum stack depth has been three and the average depth has been 1.8.
2. *Right-Recursive Grammar* This grammar first shifts all the tokens onto the stack (`a`, `+`, `b`, `+`, `c`, `+`, `d`, `+`, `e`). It then reduces `e` to *Expr*, using the second rule. It then performs a series of four reduces with the first production: `d +` *Expr* to *Expr*, `c +` *Expr* to *Expr*, `b +` *Expr* to *Expr*, and `a +` *Expr* to *Expr*. When it finishes, the maximum stack depth has been nine and the average stack depth has been 4.8.

The right-recursive grammar requires more stack space; its maximum stack depth is bounded only by the length of the expression. By contrast, the maximum stack depth with the left-recursive grammar depends on the grammar rather than the input stream.

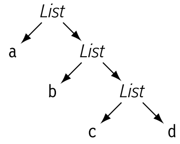
Building Lists

The same issues arise with lists of elements, such as the list of statements in a block. The compiler writer can use either left recursion or right recursion in the grammar.

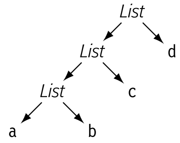
$$\begin{array}{lcl} \mathit{List} & \rightarrow & \mathit{List}\ ;\ \texttt{stmt} \\ & | & \texttt{stmt} \end{array} \qquad\qquad \begin{array}{lcl} \mathit{List} & \rightarrow & \texttt{stmt}\ ;\ \mathit{List} \\ & | & \texttt{stmt} \end{array}$$

Left-Recursive Grammar   Right-Recursive Grammar

The left-recursive grammar uses a bounded amount of stack space while the right-recursive grammar uses stack space proportional to the length of the list. For short lists, stack space is not a problem. For long lists—say a block with hundreds or thousands of statements—the difference can be dramatic. This observation suggests that the compiler writer should use the left-recursive grammar.



Right-Recursive AST



Left-Recursive AST

The problem with this approach arises when the compiler builds a data structure to represent the list. Consider a simple abstract syntax tree for a four element list: (a b c d). The AST from the right-recursive grammar reflects our intuitions about the statement list; a is first and d is last. The tree from the left-recursive grammar represents the same information; the statements are ordered correctly left to right. The nesting order, however, is somehow less intuitive than that for the right-recursive version of the AST. The code to traverse the list becomes less obvious, as well.

In many cases, the compiler writer will want to use the left-recursive grammar for its bounded stack space but build the AST that would naturally result from the right-recursive grammar. The answer is to build a list constructor that adds successive elements to the end of the list. A straightforward implementation of this idea would walk the list on each reduction, which makes the constructor take $O(n^2)$ time, where n is the length of the list.

With the right set of list constructors, the compiler writer can arrange to build the right-recursive AST from the left-recursive grammar. Consider the following syntax-driven framework:

$$\begin{array}{lcll} \mathit{List} & \rightarrow & \mathit{List}' & \{\ \texttt{\$\$} \leftarrow \texttt{RemoveListHeader()}\ ;\ \} \\ \mathit{List}' & \rightarrow & \mathit{List}'\ ;\ \texttt{stmt} & \{\ \texttt{AddToEnd(\$1,\$2)}\ ;\ \} \\ & | & \texttt{stmt} & \{\ \texttt{AddToEnd(MakeListHeader(),\$1)}\ ;\ \} \end{array}$$

The framework uses three helper functions.

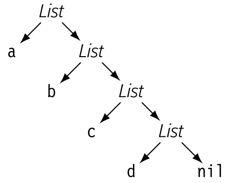
`MakeListHeader()` builds a header node that contains pointers to the start and end of a list. It returns a pointer to the header node.

`RemoveListHeader`(x) takes as input a header node x. It returns x's start-of-list pointer and discards the header node.

`AddToEnd`(x, y) takes as input a header node x and an item y. It creates a new *List* node and makes y its left child and `nil` its right child. It then uses x's end-of-list pointer to add the new *List* node to the end of the list. Finally, it returns x.

We developed this framework for an ILOC parser written in `bison`. The original right-recursive version overflowed the parse stack on inputs with more than 64,000 operations.

Each of these functions can be implemented so that it uses $\mathbf{O}(1)$ time. They work together to build and maintain both the header node and the list. The framework adds the production *List* → *List'* to create a point in the computation where it can discard the header node. The AST that it builds differs slightly from the one shown earlier; as shown in the margin, it always has a `nil` as the right child of the final *List* node.



AST Built by the Framework

5.7.2 Harder Problems in Type Inference

Strongly typed, statically checked languages can help the programmer produce valid programs by detecting large classes of erroneous programs. The same features that expose errors can improve the compiler's ability to generate efficient code for a program by (1) eliminating runtime checks or (2) exposing situations where the compiler can specialize code for some construct to eliminate cases that cannot occur at runtime. These advantages account, in part, for the growing role of type systems in programming languages.

Our examples, however, make assumptions that do not hold in all programming languages. For example, we assumed that variables and procedures are declared—the programmer writes down a concise and binding specification for each name. Varying these assumptions can radically change the nature of both the type-checking problem and the strategies that the compiler can use to implement the language.

Some programming languages either omit declarations or treat them as optional information. PYTHON and SCHEME programs lack declarations for variables. SMALLTALK programs declare classes, but an object's class is determined only when the program instantiates that object. Languages that support separate compilation—compiling procedures independently and combining them at link time to form a program—may not require declarations for independently compiled procedures.

In the absence of declarations, type checking is harder because the compiler must rely on contextual clues to determine the appropriate type for each name. For example, if the compiler sees an array reference `a[i]`, that usage might constrain the type of `i`. The language might allow only integer subscripts; alternatively, it might allow any type that can be converted to an integer.

Typing rules are specified by the language definition. The specific details of those rules determine how difficult it is to infer a type for each variable. This, in turn, has a direct effect on the strategies that a compiler can use to implement the language.

Type-Consistent Uses and Constant Function Types

Consider a declaration-free language that requires consistent uses for variables and functions. The compiler can assign a general type to each name and narrow that type by examining uses of the name in context. For example, the statement `a` ← `b` × `3.14159` suggests that `a` and `b` are numbers and that `a` must have a type that allows it to hold a decimal number. If `b` also appears in contexts where an integer is expected, such as an array reference `c[b]`, then the compiler must choose between a decimal number (for `b` × `3.14159`) and an integer (for `c[b]`). With either choice, one of the uses will need a conversion.

If functions have return types that are both known and constant—that is, a function *fee* always returns the same type—then the compiler can solve the type inference problem with an iterative fixed-point algorithm operating over a lattice of types.

Type-Consistent Uses and Unknown Function Types

Map can also handle functions with multiple arguments. To do so, it takes multiple argument lists and treats them as lists of arguments, in order.

If the type of a function varies with the function's arguments, then the problem of type inference becomes more complex. This situation arises in SCHEME, for example, where the library function *map* takes as arguments a function and a list. It returns the result of applying the function argument to each element of the list. That is, if the argument function takes type α to β, then *map* takes a list of α to a list of β. We would write its type signature as

$$map\colon (\alpha \rightarrow \beta) \times \text{list of } \alpha \rightarrow \text{list of } \beta$$

Since *map*'s return type depends on the types of its arguments, a property known as parametric polymorphism, the inference rules must include equations over the space of types. (With known, constant return types, functions return values in the space of types.) With this addition, a simple iterative fixed-point approach to type inference is not sufficient.

The classic approach to checking these more complex systems relies on unification, although clever type-system design and type representations can enable use of simpler or more efficient techniques.

Dynamic Changes in Type

If a variable's type can change during execution, other strategies may be required to discover where type changes occur and to infer appropriate types. In principle, a compiler can rename the variables so that each definition site corresponds to a unique name. It can then infer types for those names based on the context provided by the operation that defines each name.

To infer types successfully, such a system would need to handle points in the code where distinct definitions must merge due to the convergence of different control-flow paths, as with ϕ-functions in static single assignment form (see Sections 4.6.2 and 9.3). If the language includes parametric polymorphism, the type-inference mechanism must handle it, as well.

The classic approach to implementing a language with dynamically changing types is to fall back on interpretation. LISP, SCHEME, SMALLTALK, and APL all face this challenge. The standard implementation practice for these languages involves interpreting the operators, tagging the data with their types, and checking for type errors at runtime.

In APL, the expression `a×b` can multiply integers the first time it executes and multiply multidimensional arrays of floating-point numbers the next time. This feature led to a body of research on check elimination and check motion. The best APL systems avoided many of the checks that a naive interpreter would need.

5.7.3 **Relative Offsets and Cache Performance**

The widespread use of cache memories has subtle implications for the layout of variables in memory. If two values are used in proximity in the code, the compiler would like to ensure that they can reside in the cache at the same time. This can be accomplished in two ways. In the best situation, the two values would share a single cache block, to guarantee that the values are always fetched into cache together. If they cannot share a cache block, the compiler would like to ensure that the two variables map to different cache lines. The compiler can achieve this by controlling the distance between their addresses.

If we consider just two variables, controlling the distance between them seems manageable. When all the active variables are considered, however, the problem of optimal arrangement for a cache is NP-complete. Most

A PRIMER ON CACHE MEMORIES

One technique that architects use to bridge the gap between processor speed and memory speed is the use of *cache memories*. A cache is a small, fast memory placed between the processor and main memory. The cache is divided into a set of equal-sized *frames*. Each frame has a *tag* that holds enough of the main-memory address to identify the contents of the frame.

The hardware automatically maps memory locations to cache frames. The simplest mapping, used in a direct-mapped cache, computes the cache address as the main memory address modulo the size of the cache. This partitions the memory into a linear set of blocks, each the size of a cache frame. A *line* is a memory block that maps to a frame. At any point in time, each cache frame holds a copy of the data from one of its blocks. Its tag field holds the address in memory where that data normally resides.

On each read from memory, the hardware first checks to see if the requested word is already in its cache frame. If so, it returns the requested bytes to the processor. If not, (1) the block currently in the frame is evicted, (2) the requested block is fetched into the cache, and (3) the requested bytes are returned to the processor.

Some caches use more complex mappings. A set-associative cache uses multiple frames per cache line, typically two or four frames per line. A fully associative cache can place any block in any frame. Both of these schemes use an associative search over the tags to determine if a block is in the cache. Associative schemes use a policy to determine which block to evict. Common schemes are random replacement and least-recently-used replacement.

In practice, the effective memory speed is determined by memory bandwidth, cache block length, the ratio of cache speed to memory speed, and the ratio of cache hits to cache misses. From the compiler's perspective, the first three are fixed. Compiler-based efforts to improve memory performance focus on increasing the ratio of cache hits to cache misses.

Some architectures provide instructions for a program to give the cache hints as to when specific blocks should be brought into memory (*prefetched*) and when they can be discarded (*flushed*).

variables have interactions with many other variables; this creates a web of relationships that the compiler may not be able to satisfy concurrently. If we consider a loop that uses several large arrays, the problem of arranging mutual noninterference becomes even worse. If the compiler can discover the relationship between the various array references in the loop, it can add padding between the arrays to increase the likelihood that the references hit different cache lines and, thus, do not interfere with each other.

As we saw previously, the mapping of the program's virtual address space to the hardware's physical address space need not preserve the distance between specific variables. Carrying this thought to its logical conclusion, the reader should ask how the compiler can ensure anything about relative offsets that are larger than the size of a virtual-memory page. The processor's cache may use either virtual addresses or physical addresses in its tag fields. A virtually addressed cache preserves the distance between values in the virtual space; the compiler may force noninterference between large objects. With a physically addressed cache, the distance between two locations in different pages is determined by the page map (unless cache size $\leq$ page size). Thus, the compiler's decisions about memory layout have little, if any, effect, except within a single page. In this situation, the compiler should focus on placing objects that are referenced together into the same page and, if possible, the same cache line.

5.8 SUMMARY AND PERSPECTIVE

The real work of compilation is translation: mapping constructs in the source language to operations on the target machine. The compiler's front end builds an initial model of the program: an IR representation and a set of ancillary structures. This chapter explored syntax-driven translation, a mechanism that lets the compiler writer specify actions to be performed when the front end recognizes specific syntactic constructs. The compiler writer ties those actions to grammar productions; the compiler executes them when it recognizes the production.

Formal techniques have automated much of scanner and parser construction. In translation, most compilers rely on the ad-hoc techniques of syntax-driven translation. While researchers have developed more formal techniques, such as attribute grammar systems, those systems have not been widely adopted. The syntax-driven techniques are largely ad-hoc; it takes some practice for a compiler writer to use them effectively. This chapter captures some of that experience.

We suspect that attribute grammar systems have failed to win an audience because of the lack of a widely available, well-implemented, easy-to-use system.

`yacc` and `bison` won the day not because they are elegant, but because they were distributed with UNIX and they worked.

To perform translation, the compiler must build up a base of knowledge that is deeper than the syntax. It must use the language's type system to infer a type for each value that the program computes and use that information to drive both error detection and automatic type conversions. Finally, the compiler must compute a storage layout for the code that it sees; that storage layout must be consistent with and compatible with the results of other compilations of related code.

CHAPTER NOTES

The material in this chapter is an amalgam of accumulated knowledge drawn from practices that began in the late 1950s and early 1960s.

The concepts behind syntax-driven translation have always been a part of the development of real parsers. Irons, describing an early ALGOL-60 compiler, clearly lays out the need to separate a parser's actions from the description of its syntax [214]; he describes the basic ideas behind syntax-driven translation. The same basic ideas were undoubtedly used in contemporary operator precedence parsers.

The specific notation used to describe syntax-driven actions was introduced by Johnson in the `yacc` system [216]. This notation has been carried forward into many more recent systems, including the Gnu project's `bison` parser generator.

Type systems have been an integral part of programming languages since the original FORTRAN compiler. While the first type systems reflected the resources of the underlying machine, deeper levels of abstraction soon appeared in type systems for languages such as ALGOL 68 and SIMULA-67. The theory of type systems has been actively studied for decades, producing a string of languages that embodied important principles. These include RUSSELL [49] (parametric polymorphism), CLU [256] (abstract data types), SMALLTALK [172] (subtyping through inheritance), and ML [274] (thorough and complete treatment of types as first-class objects). Cardelli has written an excellent overview of type systems [76]. The APL community produced a series of classic papers that dealt with techniques to eliminate runtime checks [1,38,273,361].

Most of the material on storage layout has developed as part of programming language specifications. Column-major order for arrays appeared in early FORTRAN systems [27,28] and was codified in the FORTRAN 66 standard. Row-major order has been used since the 1950s.

EXERCISES

Section 5.3

1. Consider the problem of adding syntax-driven actions to an LL(1) parser generator. How would you modify the LL(1) skeleton parser to include user-defined actions for productions?
2. Consider the following simple grammar, which describes the language of strings over the words `one`, `two`, and `three`.

$$
\begin{array}{lcl}
\mathit{Goal} & \rightarrow & \mathit{List} \\
\mathit{List} & \rightarrow & \mathit{List}\ \ \mathit{Word} \\
 & | & \mathit{Word} \\
\mathit{Word} & \rightarrow & \underline{\texttt{one}} \\
 & | & \underline{\texttt{two}} \\
 & | & \underline{\texttt{three}}
\end{array}
$$

Each word has a weight: <u>`one`</u> = 1, <u>`two`</u> = 2, and <u>`three`</u> = 3.

Write a set of syntax-driven translation rules that computes two measures of the input sentence: the sum of the weights of all the words and the number of times that each word occurs (its frequency count).

On a reduction to the *Goal* symbol, the framework should print out both the sum and the frequency counts.

Section 5.4

3. The "define-before-use" rule requires that each variable used in a procedure be declared before it is used in the text. Sketch a simple syntax-driven translation scheme for checking that a procedure conforms to this rule. (**Hint:** Use a map.)

   Is this problem any easier if the language requires that all declaration statements precede any executable statement?

4. For the program shown in Fig. 5.9, draw the complete set of symbol tables and show the search paths that the compiler should construct for each of `Main`, `Fee`, `Fie`, `Foe`, and `Fum`.

5. When the compiler encounters a structure definition, it must treat the interior of that definition as a new scope. (Names defined as structure elements obscure identical names in the surrounding scope.)

   a. If the compiler represents these scopes with independent tables, how might it link them into the search path?

   b. Can the compiler use *qualified names* (e.g., `x.next`) to represent these names?

6. The compiler must store information in the IR version of the program that allows it to find the symbol table entry for each name. Among the options open to the compiler writer are pointers to the original character strings, pointers to the symbol table entries, and a pair that contains a pointer to the specific table and an offset within it.

   What are the advantages and disadvantages of each of these representations for a name? How would you choose to represent names in a compiler that you designed from scratch?

```
print "enter a number:";
n = input();

if n < 0:
    v = "string";
else:
    v = 23;

print v;
```

(a) Original PYTHON Program

```
print "enter a number:";
n = input();

if n < 0:
    v = "string";
    print v;
else:
    v = 23;
    print v;
```

(b) Modified PYTHON Program

■ **FIGURE 5.19** Code for Exercise 7.

Section 5.5

7. Fig. 5.19 shows two versions of a PYTHON program. PYTHON is dynamically typed, so the type of v is determined, at runtime, by assignments to v. You are developing a tool that tries to perform static checking for PYTHON programs.

   a. What is the type of v in the print statement in panel (a)?

   b. What is the type of v in each print statement in panel (b)?

   c. Would there be a difference in the type signature that your tool derived for `print` for the two versions of the code?

   d. What problems arise if you translate each of these programs to C?

8. Assume that your compiler uses a framework that combines the work shown in Figs. 5.5 and 5.13. Augment that framework so that it converts correctly between integer and floating-point values based on the types of each subexpression. To provide a concrete IR, use ILOC as described in Appendix A.

Section 5.6

9. The compiler must assign each variable an offset within its data area. Assume that the algorithm receives as input a list of variables, their lengths, and their alignment restrictions, such as:

   $\langle \texttt{a}, 4, 4\rangle$, $\langle \texttt{b}, 3, 1\rangle$, $\langle \texttt{c}, 8, 8\rangle$, $\langle \texttt{d}, 4, 4\rangle$, $\langle \texttt{e}, 1, 4\rangle$, $\langle \texttt{f}, 8, 16\rangle$, $\langle \texttt{g}, 1, 1\rangle$.

   The algorithm should produce, as output, a list of the variables and their offsets in the data area. The algorithm's goal is to minimize unused, or wasted, space.

   a. Write down a concise algorithm to perform layout that minimizes wasted space.

   b. Apply your algorithm to the example list above and to two other lists that you design to demonstrate the problems that can arise in storage layout.

   c. What is the complexity of your algorithm?

10. Given an array of integers with dimensions `A[0:99,0:89,0:109]`, how many words of memory are required to represent `A` in row-major order and in a set of indirection vectors? Assume that integers and pointers are the same size.

11. For each of the following categories of variables, state where in memory the compiler might allocate space for such a variable. Possible answers include registers, activation records, static data areas (with different visibilities), and the runtime heap.

    a. A scalar ambiguous local variable
    b. A scalar unambiguous local variable
    c. A global variable
    d. A dynamically allocated global variable
    e. A formal parameter of the procedure
    f. A compiler-generated temporary value
    g. A local variable with size determined at runtime

Section 5.7

12. For the left-recursive and right-recursive addition grammars given in Section 5.7.1, show all of the stack states that occur in parsing the expression `a + b + c + d + e + f`. Compute the maximum stack depth and the average stack depth with each grammar. What is the asymptotic value for average stack depth with each grammar?

This page intentionally left blank

Chapter 6

Implementing Procedures

ABSTRACT

Procedures play a critical role in the development of software systems. They provide abstractions for control flow and naming. They provide basic information hiding. They are the building block on which systems provide interfaces. They are one of the principal forms of abstraction in Algol-like languages; object-oriented languages rely on procedures to implement their methods or code members.

This chapter takes an in-depth look at the implementation of procedures and procedure calls, from the perspective of a compiler writer. It highlights the implementation similarities and differences between Algol-like languages and object-oriented languages. The Advanced Topics section presents a brief introduction to the algorithms used to manage the runtime heap.

KEYWORDS

Procedure Calls, Parameter Binding, Linkage Conventions

6.1 INTRODUCTION

The procedure is one of the central abstractions in most programming languages. Procedures create a controlled execution environment; each procedure has its own private storage. Procedures help define interfaces between system components; cross-component interactions are usually structured through procedure calls. Finally, procedures are the basic unit of work for most compilers. A typical compiler processes a collection of procedures and emits code for them. The code produced by these separate compilations must link and execute correctly with code compiled at other times.

Procedures play a critial role in separate compilation, which allows software developers to build large software systems. If the compiler needed the entire text of a program for each compilation, large software systems would be untenable. Imagine recompiling a multimillion line application for each editing change made during development! Thus, procedures play as critical a role in system design and engineering as they do in language design and compiler implementation. This chapter focuses on how compilers implement procedures and methods.

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00012-7

Conceptual Roadmap

To translate a source-language program into executable code, the compiler must map all of the source-language constructs that the program uses into operations and data structures on the target processor. The compiler needs a strategy for each of the abstractions supported by the source language. These strategies include both algorithms and data structures that are embedded into the executable code.

To implement these strategies, the compiler writer must put in place compile-time computations and data structures, mechanisms to pass information from compile time to runtime, and code in the executable to implement the runtime side of the strategy. All these elements combine to create the desired behavior when the application program runs.

This chapter explains the techniques that compiler writers use to implement procedures and procedure calls. Specifically, it examines the implementation of control, of naming, and of the call interface. These abstractions encapsulate many of the features that make programming languages usable and that enable construction of large-scale systems.

A Few Words About Time

This chapter deals with both compile time and runtime issues. The compiler plans the desired runtime behavior. It emits code to implement that behavior—typically, the standard call and return mechanisms. To understand how calls work, the reader should pay careful attention to the interplay between compile-time planning and runtime behavior.

Caller
In a procedure call, the procedure that initiates the call is termed the *caller*.

Callee
In a procedure call, the procedure that is called is termed the *callee*.

For example, the compiler must plan for the storage needs of each procedure. As it analyzes the procedure's code, the compiler builds a storage map that includes the type and size of each value computed in the procedure. It assigns each value a runtime location. Using the map, the compiler generates code that will, at runtime, allocate, initialize, and free storage for those values. The code to implement these decisions is spread across the code for the caller and the callee.

Similarly, the compiler determines what actions must be taken at each call site to preserve and protect the environment of the calling procedure; it then emits code to enact those decisions at runtime. Such decisions include saving and restoring registers, evaluating and binding parameters, and maintaining data structures that ensure correct and appropriate access to values in surrounding scopes. Again, the compiler analyzes, plans, and emits code. At runtime, that code creates the desired behavior.

Overview

Procedures are the building blocks of programs. They create a known and controlled execution environment. A procedure executes when it is invoked, or called, by another procedure. That call creates and initializes procedure-local storage. It protects the caller's environment, establishes the callee's environment, and creates any linkages between those environments specified by the call and the language. The callee may return a value to its caller, in which case the procedure is termed a *function*. The interface between procedures lets programmers develop and test parts of a program in isolation; the separation between procedures provides some insulation against problems in other procedures.

Procedures shape the way that programmers develop software and that compilers translate programs. Three critical abstractions that procedures provide allow the construction of nontrivial programs.

1. *The Call* Procedure calls provide an orderly transfer of control between procedures. The call mechanism provides a standard way to invoke a procedure and map a set of arguments, or parameters, from the caller's name space to the callee's name space. After the callee completes, control returns to the caller, at the point immediately after the call. Most languages allow a procedure to return one or more values to the caller.

   Languages and implementations define a standard set of actions required to invoke a procedure, sometimes referred to as a *calling sequence*. Standardization of calling sequences, in the form of a *linkage convention*, lets the programmer invoke code written and compiled by other people and other compilers at other times.

Linkage convention
an agreement between the compiler and operating system that defines the actions taken to call a procedure or function

2. *Name Space* In most languages, a procedure call creates a new and protected name space for the callee (see Section 5.4). The programmer can declare new names, such as variables and labels, inside the procedure without concern for the surrounding context. Those local declarations obscure any previously defined items with the same names. The programmer can use formal parameters in the procedure to map values and variables from the caller's name space into the callee's name space. Because the procedure has a known and separate name space, it can function correctly and consistently when called from different contexts.

   Executing a call instantiates the callee's name space. The call must create storage for the objects declared by the callee. This allocation must be both automatic and efficient.

Actual parameter
A value or variable passed as a parameter at a call site is an *actual parameter* of the call.

Formal parameter
A name declared as a parameter of some procedure *p* is a *formal parameter* of *p*.

3. *External Interface* Procedures form critical interfaces among the parts of large software systems. The linkage convention defines rules that map names to values and locations, that preserve the caller's runtime

environment and create the callee's environment, and that transfer control between caller and callee.

The linkage convention creates a standard way to invoke code written by other people and in other languages. Uniform calling sequences enable the construction of large software systems by making possible the use of libraries and system calls. Without a linkage convention, both the programmer and the compiler would need to understand the callee's code at each procedure call.

Thus, the procedure is, in many ways, the fundamental abstraction that underlies Algol-like languages (ALLs). It is an elaborate facade created collaboratively by the compiler, the underlying hardware, and the operating system. Procedures create named variables and map them to virtual addresses; the operating system and the hardware map those virtual addresses to physical addresses. Procedures establish rules for visibility and addressability of names; the hardware loads and stores values from addresses computed by the compiled code. Procedures let us decompose large software systems into components; linkers and loaders knit these components together into an executable; the hardware runs that code by simply advancing its program counter and following branches and jumps.

Implementing the linkage convention requires coordination between the compiler and the surrounding system, as well as between compilations. The compiler must arrange for the generated code to use the interfaces provided by the operating system to handle input and output, to manage memory allocation and deallocation, and to communicate with other processes. The compiler must lay out memory for each procedure and encode that knowledge into the generated code. Since the different components of the program may be compiled at different times, with different compilers, and in isolation from each other, all of these interactions must be standardized and uniformly applied.

To complicate matters, the user wants the optimizer to specialize the generated code to its runtime context so that it runs as quickly as practical (see Chapters 8–10).

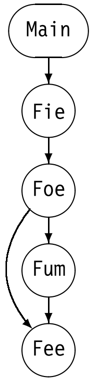
This chapter focuses on the procedure as an abstraction and the mechanisms that the compiler uses to establish its control abstraction, name space, and interface to the outside world.

6.2 BACKGROUND

In ALLs, procedures have a simple call/return discipline. A call transfers control from a call site in the caller to the start of the callee; when the callee exits, control returns to the point in the caller immediately after the call. If the callee, in turn, invokes other procedures, they follow the same protocol, with a call and a return. Fig. 6.1 shows the call graph and *execution history* for the PASCAL program from Fig. 5.9.

```
program Main(input, output);
   var x,y,z: integer;
   procedure Fee;
      var x: integer;
      begin { Fee }
         x := 1;
         y := x * 2 + 1
      end;
   procedure Fie;
      var y: real;
      procedure Foe;
         var z: real;
            procedure Fum;
               var y: real;
               begin { Fum }
                  x := 1.25 * z;
                  Fee;
                  writeln('x = ',x)
               end;
         begin { Foe }
            z := 1;
            Fee;
            Fum
         end;
      begin { Fie }
         Foe;
         writeln('x = ',x)
      end;
   begin { Main }
      x := 0;
      Fie
   end.
```

(a) Example PASCAL Program



(b) Call Graph

```
 1.  Main calls Fie
 2.  Fie calls Foe
 3.  Foe calls Fee
 4.  Fee returns to Foe
 5.  Foe calls Fum
 6.  Fum calls Fee
 7.  Fee returns to Fum
 8.  Fum returns to Foe
 9.  Foe returns to Fie
10.  Fie returns to Main
```

(c) Execution History

■ **FIGURE 6.1** Nonrecursive PASCAL Program and Its Execution History.

The call graph in panel (b) shows all of the potential calls in the program. Executing Main can produce two distinct calls to Fee: one from Foe and the other from Fum. The execution history shows that both calls actually occur. Each call creates a distinct instance, or *activation*, of Fee. By the time that Fum is called, the first instance of Fee is no longer active. It was created by the call from Foe (event 3 in the execution history), and destroyed after control returned to Foe (event 4). When Fum calls Fee (event 6), the call creates a new activation of Fee. The return from Fee to Fum ends that activation and makes its state inaccessible.

Activation
A call to a procedure *activates* it; thus, we call an instance of its execution an *activation*.

`Fee`, `Foe`, `Fie`, and `Main` are active when the program executes the assignment `x := 1` in the first call to `Fee`. They all lie on a path in the call graph from `Main` to `Fee`. Similarly, `Fee`, `Fum`, `Foe`, `Fie`, and `Main` are active when the program makes its second call to `Fee`; they lie on another path from `Main` to `Fee`. The call and return mechanism ensures that, at any point during execution, the procedure activations instantiate some rooted path through the call graph.

Diverge
A computation that does not terminate normally is said to *diverge*.

Return address
When *p* calls *q*, the address in *p* where execution should continue after *q*'s return is called its *return address*.

When the compiler generates code for calls and returns, that code must preserve enough state information to ensure correct operation on a return. When `Foe` calls `Fum`, the code must record `Foe`'s state, including the address in `Foe` to which `Fum` should return control—its *return address*. Unless `Fum` diverges, control will return to `Foe` and continue execution in `Foe`. The call mechanism preserves enough state to allow that to happen.

The call and return behavior of ALLs can be modeled with a first-in, first-out stack. Thus, the runtime implementations of many programming languages use a stack to hold the state of each active procedure. When `Fie` calls `Foe`, it pushes the new state for `Foe` onto the stack; `Foe` executes from that state. When `Foe` returns, it pops that state off the stack and exposes the state for `Fie`, as it was when the call was made.

The stack mechanism handles recursion. In effect, it unrolls the cyclic path through the call graph and creates a distinct activation and state for each call to a procedure. As long as the recursion terminates, this path will be finite and the stack will correctly capture the program's behavior.

To make this concrete, consider the following recursive factorial computation, written in SCHEME:

```
(define (fact k)
    (cond
        [(<= k 1) 1]
        [else (* (fact (sub1 k)) k)]
    ))
```

A call to `(fact 5)` generates a series of recursive calls: `(fact 5)` calls `(fact 4)` calls `(fact 3)` calls `(fact 2)` calls `(fact 1)`. At that point, the `cond` statement executes the clause for `(<= k 1)`, ending the recursion. The recursion unwinds in the reverse order. The call to `(fact 1)` returns the value 1 to `(fact 2)`. It, in turn, returns the value 2 to `(fact 3)`, which returns 6 to `(fact 4)`. Finally, `(fact 4)` returns 24 to `(fact 5)`, which multiplies 24 times 5 to return the answer 120. The recursive program exhibits last-in, first-out behavior, so the stack mechanism correctly tracks all of the return addresses.

Procedure Calls in Object-Oriented Languages

From the perspective of procedure calls, object-oriented languages (OOLs) are similar to ALLs. The primary differences between procedure calls in an OOL and an ALL lie in the mechanism used to name the callee and in the mechanisms used to locate the callee at runtime.

In an OOL, the call is made relative to an object. The compiler knows the names of the receiver and the method. From those facts, it must determine the receiver's class and resolve the method name to the intended implementation. If the compiler can resolve that name-to-implementation binding at compile time, it can emit code that looks much like a standard call in an ALL. If, however, it cannot resolve that binding until runtime, then the compiler must emit code that uses runtime data structures to resolve it.

To support runtime name resolution, the compiler must emit code that builds runtime data structures to instantiate a model of the inheritance hierarchy. Fig. 5.17 showed a simple example.

More Complex Interprocedural Control Flow

Following the lambda calculus, some programming languages allow a program to encapsulate a procedure and its runtime context into an object called a *closure*. When the closure is invoked, the procedure executes in the encapsulated runtime context. A simple stack is inadequate to implement this functionality. Instead, the control information must be saved in a more general structure that can represent the procedure and its context. Similar problems arise if the language allows references to local variables that outlast a procedure's activation.

Closure
a procedure and the runtime context that defines its free variables

SECTION REVIEW

In Algol-like languages, procedures are invoked with a call and they terminate in a return. The compiler must arrange for the call to record and preserve the caller's state and for the return to find and restore the caller's state. Using a stack to hold the state for procedure activations captures the last-in, first-out behavior of the call/return mechanism.

OOLs and languages that support closure build on this basic infrastructure to produce more complex control flow. OOLs use data-centric naming schemes to map names to method implementations. Closures break the relationship between the lifetime of a scope and the call that creates it; these schemes require more complex implementations.

REVIEW QUESTIONS

1. Many programming languages include a direct transfer of control, a so-called `goto`. Compare and contrast a procedure call and a `goto`.
2. Many languages make use of libraries of precompiled code. What facts does the compiler need about the library code so that it can ensure correct behavior? How does your favorite language provide it to the compiler?

6.3 RUNTIME SUPPORT FOR NAMING

The overall goal is to resolve a compile-time name into a runtime virtual address.

To support the name spaces that occur in source-language programs, the compiler must be able to map a given reference into its runtime virtual address. The mechanism has two parts: a compile-time capability to resolve the name to a specific runtime entity and a runtime method to locate and access that entity.

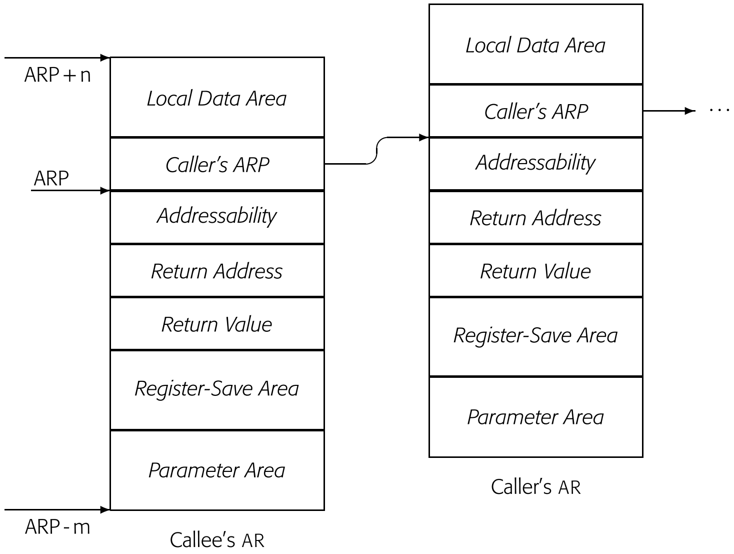
The description of these mechanisms is spread over several chapters in the text. Section 5.4 describes the compile-time mechanisms used to resolve a source-language name to a specific runtime entity. Section 5.6 discusses how the compiler determines where in the computer's memory each entity that needs storage will reside. This section explores the runtime data-structures that the compiler must put in place to facilitate addressability and to store values. Finally, Section 7.3 enumerates how the compiler uses these mechanisms to emit code that can compute a runtime virtual address.

The runtime support for name resolution and translation occurs, primarily, in two places. The data structures are created, populated, and dismantled as part of the implementation of procedure calls. Section 6.3.1 describes the mechanisms commonly used to support the nested lexical scopes that occur in Algol-like languages (ALLs). Section 6.3.2 discusses the additional runtime structures required to support object-oriented languages (OOLs). The remainder of the implementation occurs when the compiler emits code for individual references, following the plans laid out in Section 7.3.

6.3.1 Runtime Support for Algol-Like Languages

Activation record
a region of storage set aside to hold control information and data storage associated with a single instance of a single procedure

To implement the twin abstractions of procedure calls and lexically scoped name spaces, the compiler and runtime system must establish a set of runtime data structures. The key structure involved in both control and naming is the *activation record* (AR), a private block of memory associated with a single procedure activation. Each procedure call gives rise to a new AR.



■ **FIGURE 6.2** Typical Activation Records.

- The compiler must emit code so that the caller stores the return address for the callee. The return address goes in the callee's AR.
- The compiler must map the actual parameters at the call site into the formal parameters by which they are accessible in the callee. This parameter information goes into the callee's AR.
- The compiler must create storage space for local variables declared in the callee. For the values whose lifetimes match the procedure's invocation, that storage goes into the callee's AR.
- The compiler must provide the callee with other information needed to connect it to the calling context and allow it to interact with other procedures. That data goes into the callee's AR.

Each call creates a new AR. If multiple instances of a procedure are active, each has its own AR. Thus, recursion gives rise to multiple ARs, each of which holds the local state for a different activation of the procedure.

Fig. 6.2 shows how the contents of an AR might be laid out. The *activation record pointer* (ARP) points to a fixed spot in the AR. The various fields in the AR are found at positive and negative offsets from the ARP. The ARs shown in Fig. 6.2 have a number of fields.

Activation record pointer
The compiler ensures that a pointer to the AR, the *activation record pointer*, is in a designated register.
We denote that register r_{arp}.

- The parameter area holds actual parameters from the call site, in an order determined by their order of appearance at the call site.
- The register save area contains enough space to hold the values from registers that the procedure must preserve across procedure calls.

- The return-value slot provides space to communicate data from the callee back to the caller, if needed.
- The return-address slot holds the runtime address to which control should return when the callee terminates.
- The addressability slot holds information that lets the callee access local variables in surrounding lexical scopes.

The surrounding lexical scope may be a procedure other than the caller.

- The slot at the callee's ARP preserves the caller's ARP so that the callee can restore the caller's environment when it returns.
- The local data area holds variables declared in the callee's local scope that the callee cannot keep in registers, along with other values that the compiler needs to store.

For the sake of efficiency, some of the information shown in the ARs of Fig. 6.2 may be kept in dedicated registers.

Local Storage

The AR for an activation of procedure q holds both local data and control information for that instance of q. Each time the code calls q, that call generates a new AR. Accesses to fields in an AR occur through the ARP. Thus, the ARP acts as a handle to the activation's context and state. At runtime, the code accesses the AR frequently enough that most compilers dedicate a physical register to hold the ARP of the active procedure. In ILOC, we refer to this dedicated register as r_{arp}.

The ARP points to a designated position in the AR. The central part of the AR has a static layout; all the fields have known fixed lengths. Thus, the compiled code can access those items at fixed offsets from the ARP. The two ends of the AR are reserved for storage areas whose sizes depend on specific details of the corresponding procedure; one typically holds the parameters while the other holds the local data.

Reserving Space for Local Data

When the compiler performs storage assignment, it assigns space in the local data area to automatic local variables that cannot reside in registers (see Section 5.6.3). As it assigns space to such variables, it should record the current lexical level and the value's offset from the ARP in the symbol table. This pair, a lexical level and offset, is the key to accessing the value at runtime. This pair becomes its static coordinate (see Section 7.3.1).

The input program may contain one or more variables for which the size is not known at compile time. For example, the code might compute the size of an array or read its size from external media. In such cases, the compiler can create space in the local data area for a pointer to either the actual data or a dope vector for the actual data (see Section 7.3.2).

The compiler can then arrange to allocate the space needed for the data elsewhere. It might put the data in the heap; if the ARs are on a stack, it might place the data at the top of the stack. In either scenario, the compiler can arrange to store a pointer to the storage in the space that it created in the AR, or in the appropriate slot of the dope vector. The static coordinate then leads to the slot in the AR; the code for the array access then begins with that pointer.

Initializing Variables

If the source language lets the programmer specify initial values for variables, the compiler must ensure that the initialization occurs. For a statically allocated variable, the data can be inserted directly into the appropriate locations in the static data area.

Most assemblers provide a pseudooperation to allocate and initialize static storage.

Variables stored in the AR must be initialized at runtime. Because a procedure may be invoked multiple times, the compiler must emit operations to perform the initializations on every invocation. In effect, these initializations are assignments that execute before the procedure's first statement, on each invocation.

Space for Saved Register Values

When p calls q, one of them must save the register values that p needs after the call. It may be necessary to save all the register values; on the other hand, a subset may suffice. On return to p, these saved values must be restored. The lifetime of these saved register values matches the lifetime of the callee. Thus, the compiler can save them in either the callee's AR or the caller's AR.

Caller-saves register
If the caller has responsibility to preserve the value of r_i across a call, we say that r_i is a *caller-saves register*.

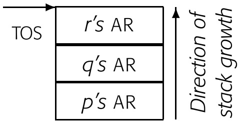
Callee-saves register
If the callee has responsibility to preserve the value of r_i across its own execution, we say that r_i is a *callee-saves register*.

If the caller saves a register before jumping to the callee, it saves the value into its own AR. Similarly, if the callee saves a register before it begins to execute, it saves the value into its own AR. Thus, the AR needs capacity to store a full set of caller-saves and callee-saves registers. Since the caller saves before the call and restores after the call, it can reuse the same space for caller-saves registers at each call.

Allocating Activation Records

When procedure p calls procedure q, the code for the call must allocate and initialize an AR for q. Because procedure calls are frequent, the compiler writer should try to keep the cost of this allocation as low as possible. In general, the compiler writer has three choices for AR allocation: stack allocation, heap allocation, and static allocation.

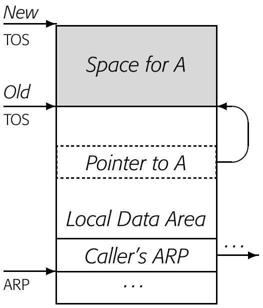
Stack Allocation of Activation Records



Stack Allocation of ARs
p calls *q* calls *r*

In many cases, the contents of an AR are only of interest during a single activation of the procedure. If local variables cannot outlive the procedure that creates them and procedure activations cannot outlive their callers, then calls and returns are balanced and follow a last-in, first-out (LIFO) discipline. A call from *p* to *q* eventually returns, and any returns that occur between the call from *p* to *q* and the return from *q* to *p* must result from calls made (either directly or indirectly) by *q*.

With these restrictions, the ARs also follow the LIFO ordering and allocation can use a simple stack. PASCAL, C, and JAVA are typically implemented with stack-allocated ARs, as shown in the margin.



Extending a Stack-Allocated AR

Stack allocation of ARs has several advantages. Allocation and deallocation are inexpensive; the code performs arithmetic on the top-of-stack (TOS) pointer. The code can extend the AR at the top of the stack by testing for stack overflow and then adjusting the TOS pointer. This scheme lets a procedure extend its AR to create space for dynamically sized local entities. The drawing in the margin shows this situation. Here, the compiler has left a slot in the local data area for a pointer to *A*, and later extended the AR to create space for *A*.

With stack allocation, the task of creating the AR can be split between the caller and the callee. The caller can create those portions of the AR whose size it knows—everything from the parameter area to the caller's ARP. The callee can then extend its own AR to create space for the local data area.

Heap Allocation of Activation Records

If the programming language allows a procedure activation to outlive its caller, the stack discipline for allocating ARs breaks down. Similarly, if a procedure can return an object, such as a closure, that includes, explicitly or implicitly, references to its local variables, stack allocation is inappropriate because it will leave behind dangling pointers. In these situations, the compiler can allocate ARs on a runtime heap (see Section 6.6). Implementations of SCHEME and ML typically use heap-allocated ARs.

A good memory allocator can keep the cost of heap allocation low. Most heap implementations do not allow an object to expand after it is allocated. This fact complicates AR allocation because it implies that either:

1. The caller must know the size of the callee's local data area. The compiler can compute that size and store it in a global constant pool. At each call, the caller can retrieve the value and include it in the calculation of the local data area's size.

2. The callee needs to allocate the local data area separately. This option, while simpler, may require a separate register to hold a pointer to the local data area.

With heap-allocated ARs, variable-size objects can be allocated as separate objects on the heap. If heap objects need explicit deallocation, then the code for procedure return must free the AR and its variable-size extensions.

Heap allocated ARs work well for closures. The closure holds a pointer to the AR for the activation that created it. The chain of caller's ARP fields will keep the rest of the state live until the closure is invoked. When the closure is discarded, any parts of that state that were live only because of the closure will become dead; an implicit deallocation scheme will recycle them.

Static Allocation of Activation Records

Leaf procedure
a procedure that contains no calls

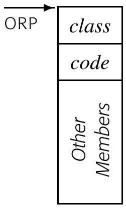
If a procedure *q* calls no other procedures, then *q* can never have multiple active invocations. We call *q* a *leaf procedure* since it terminates a path through a graph of possible procedure calls. The compiler can statically allocate ARs for leaf procedures to avoid the runtime costs of AR allocation. If the calling convention has only caller-saves registers (no callee-saves registers), then *q*'s AR needs no register save area.

If the language does not allow closures, the compiler can do better than one static AR per leaf procedure. At any point during execution, only one leaf procedure can be active. (To have two such procedures active, the first one would need to call another procedure, so it would not be a leaf.) Thus, the compiler can allocate a single static AR for use by all of the leaf procedures. The static AR must be large enough to accommodate any of the program's leaf procedures. The static variables declared in any of the leaf procedures can be laid out together in that single AR, which avoids the need for separate static ARs for each leaf procedure.

Coalescing Activation Records

If the compiler discovers a set of procedures that are always invoked in a fixed sequence, it may be able to combine their ARs. For example, if a call from *p* to *q* always results in calls to *r* and *s*, the compiler may find it profitable to allocate the ARs for *q*, *r*, and *s* at the same time. Combining ARs can save on the costs of allocation; the benefits will vary directly with allocation costs. In practice, this optimization is limited by separate compilation and the use of function-valued parameters. Both limit the compiler's ability to determine the calling relationships that actually occur at runtime.

6.3.2 Runtime Support for Object-Oriented Languages



Format of an Object Record

Just as Algol-like languages need runtime structures such as ARs to support their lexical name spaces, so too do object-oriented languages need runtime structures to support both their lexical hierarchy and their class hierarchy. Some of those structures are identical to the ones found in an ALL. For example, the control information for methods, as well as storage for method-local names, is kept in ARs. Other structures are designed to address specific needs of the OOL. For example, object lifetimes need not match the lifetime of any particular method activation, so their persistent state cannot be stored in some AR. Thus, each object needs its own object record (OR) to hold its state. The ORs of classes instantiate the inheritance hierarchy; they play a critical role in translation and execution.

Each OR begins with a member *class* that indicates the object's class, followed by a member *code* that points to the class' method vector, as shown in the margin. The remaining class-specific members follow *code*. To locate an OR, the code keeps an OR pointer (ORP) that refers to a fixed offset in the OR; the drawings will assume it points to the start of the OR.

The amount of runtime support that an OOL needs depends heavily on features of the OOL. To explain the range of possibilities, we will begin with single inheritance and an open class structure, using the example shown in Section 5.4.2. From there, we will explore the implications of a closed class structure for method lookup.

Panels (a) and (b) in Fig. 6.3 repeat the example from Chapter 5. Panel (c) shows the runtime structures that might result from instantiating both a `Point` and a `ColorPoint`. All of `Point`, `ColorPoint`, `aPoint` and `aColorPoint` have their own ORs. The OR for `class` is off-page.

The figure shows complete method vectors in each class, ordered in a prefix layout.

The OR for `aPoint` is simple. It contains a pointer to `aPoint`'s class, a pointer to `aPoint`'s method vector (from class `Point`), and space for the data members of a `Point`.

The OR for `aColorPoint` is equally simple. It contains a pointer to `aColorPoint`'s class, a pointer to its method vector (from class `ColorPoint`), and space for the data members of a `ColorPoint`. The data members are ordered in a prefix-layout scheme, following the discussion on layout for single-inheritance from Section 5.6.3 on page 255. Thus, the inherited data members, `x`, `y`, and `z`, sit at the same offsets as they would in an instance of `Point`, and the data members declared in `ColorPoint` sit below them.

The OR for a class has a similar structure. Each OR of class `class` contains a class pointer. Their code pointers, which are not shown, would lead to

```
Class Point {
    public int x, y;
    private int z;
    public void draw() {...};
    public void move() {...};
}
```

```
Class ColorPoint extends Point {
    private Color c;
    public void draw() {...};
    public void setc( Color x )
        { this.c = x };
}
```

(a) Class Definitions

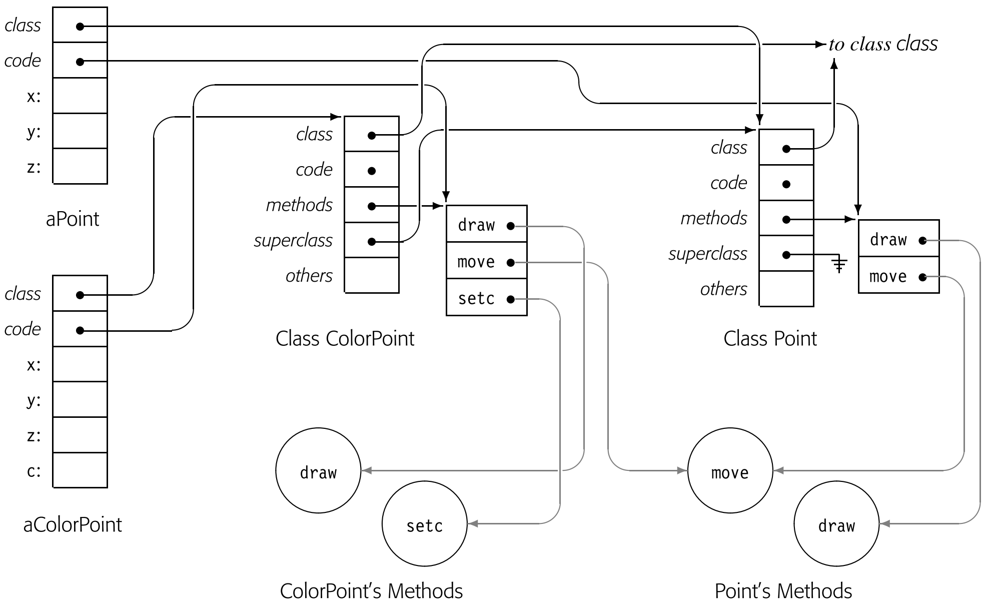
Class: Point
Superclass: *none*

x	int	public
y	int	public
z	int	private
draw	void()	public
move	void()	public

Class: ColorPoint
Superclass: Point

c	Color	private
draw	void()	public
setc	void()	public

(b) Corresponding Scope Tables



(c) Object Records for Point, ColorPoint, aPoint, and aColorPoint

■ **FIGURE 6.3** Object Layout, Linking, and Inheritance.

the method vector in class class. The class ORs show the method vector and superclass pointers. Undoubtedly, the OR's of classes would contain additional class-specific members.

Allocation of Object Records

ORs are allocated explicitly when an object is created and deallocated when the object is no longer reachable. Most ORs are heap allocated, because object lifetimes are not typically correlated to a single activation of some method. If an object's lifetime is bounded by some method's activation, then its OR can be stored inside the AR created for that method.

Analysis can reveal cases where a heap-allocated OR may be, instead, placed inside some method's AR. This transformation can reduce both allocation overhead and the cost of heap management.

Method Invocation

How does the compiler generate code to invoke a method such as draw? Methods are always invoked relative to some receiver object, say an instance of aColorPoint. For the invocation to be legal, aColorPoint must be visible at the point of the call so that the compiler can discover how to find aColorPoint with a symbol-table lookup. The compiler first looks in the method's lexical hierarchy, then in the class hierarchy, and, finally, in the global scope. That lookup provides enough information to let the compiler emit code to obtain aColorPoint's ORP.

Once the compiler has emitted code to obtain the ORP, it can find and retrieve the code-vector pointer at a fixed offset from the ORP. The pointer to draw is at a fixed offset from the start of the code vector. (As drawn, that offset is zero.) The compiler can use draw's address in a standard procedure call, with one twist—the call passes aColorPoint's ORP as draw's implicit first parameter, which is bound to a language-specified name such as this or self.

Careful construction of the runtime data structures allows this strategy to work. The code pointer in aColorPoint's OR points to the method vector from ColorPoint rather than the one from Point. Thus, invoking draw relative to aColorPoint calls ColorPoint.draw. Invoking draw relative to aPoint would call Point.draw. Finally, casting aColorPoint to be a Point should cause it to use Point's method vector and call Point.draw.

The example in Fig. 6.3 shows each class with a complete method vector. Thus, ColorPoint's method vector has pointers to ColorPoint.draw, Point.move, and ColorPoint.setc. The vector embodies the result of resolving those names through the inheritance hierarchy. This scheme achieves the intended result—an object of class x invokes the implementation of a method that is visible from inside class x.

As an alternative, the compiler can represent only ColorPoint's local methods in its class method vector. A method name would resolve to a coordinate that told the compiler how far up the superclass chain it must run to find the code pointer. The compiler would emit code to chase up the superclass chain to the correct level and then retrieve the code pointer. Complete method vectors save runtime at the expense of a minor increase in runtime space.

Static Versus Dynamic Dispatch

The discussion so far suggests that every method call requires a lookup through the receiver's OR to locate the method's implementation. In a language with a closed class structure, the compiler can resolve the method name to a specific implementation at compile time and generate a direct call. In C++, for example, the compiler can resolve any method to a concrete implementation at compile time, unless the method is declared as a *virtual method*—meaning, essentially, that the programmer wants to locate the implementation relative to the receiver's class.

Dispatch
The process of calling a method is often called *dispatch*, a term derived from the message-passing model of OOLs such as SMALLTALK.

With a virtual method, dispatch locates an implementation using the appropriate method vector. The compiler emits code to follow the path from OR to class method vector to code pointer, a process often called *dynamic dispatch*. If, however, the C++ compiler can prove that some virtual method call has a known invariant receiver class, it can generate a direct call, sometimes called *static dispatch*.

Languages with open class structures may need to rely on dynamic dispatch. If the class structure can change at runtime, the compiler cannot resolve method names to implementations; instead, it must defer this process to runtime. The techniques used to address this problem range from recomputing method vectors when the class hierarchy changes, to recompiling affected classes with a JIT, to runtime name resolution and a runtime search in the class hierarchy.

- If the class hierarchy changes infrequently, the implementation may simply rebuild method vectors for the affected classes after each change. In this scheme, the runtime system must traverse the superclass hierarchy to locate method implementations when it builds subclass method vectors. Changes in class structure invalidate any JIT-compiled code for related methods (see Chapter 14).
- If the class hierarchy changes often, the implementation may keep incomplete method vectors in each class—that is, a class' method vector only holds pointers to methods that are local to that class. In this scheme,

METHOD CACHES

To support an open class hierarchy, the compiler may need to produce a search key for each method name and maintain a runtime-searchable map of ⟨*class*, *key*⟩ pairs to implementations. In such a scheme, method dispatch includes a search, by key, through tables in the class hierarchy.

To improve method lookup in this situation, the runtime system can implement a *method cache*—a software analog of a processor's data cache. Each method cache entry consists of a key, a class, and a code pointer.
A dynamic dispatch begins with a lookup in the method cache; if it finds the ⟨*class*, *key*⟩ pair then it returns the code pointer. If the lookup fails, the dispatch searches up the superclass chain from the receiver's class. When it finds the method, it caches that result and returns the code pointer.

Of course, creating the new entry may force eviction of some other cache entry. Standard cache replacement policies, such as least recently used or round robin, can select the method to evict. Larger caches retain more information, but require more memory and may take longer to search.

To capture type locality at individual calls, some systems use an *inline method cache*, a single-entry cache located at each call site. The cache stores the receiver's class and the code pointer from the last execution of that site. If the current receiver's class matches the cached class, the call uses the cached code pointer. If the current receiver's class does not match the cached class, a full lookup is performed; that lookup records its result in the cache.

A change to the class hierarchy must invalidate the affected entries in the method caches. The implementation could clear the method cache and any inline caches; they will refill as the code executes. An alternative solution that works well with inline caches is to generate a new tag for each affected class. The old tags will not match. New lookups will find the new methods. The outdated entries will eventually be evicted.

a call to a superclass method triggers a runtime search in the class hierarchy for the first method of that name.

Keys might be kept in a table local to the class; alternatively, the method vector could consist of ⟨*key*,*address*⟩ pairs.

In either case, the language runtime will need lookup tables of method names—either source-level names or search keys derived from those names. Conceptually, each class needs a small dictionary. Runtime name resolution looks up the method name in the hierarchy, in a manner analogous to the chain of symbol tables described in Section 4.5.1.

OOL implementations use two general strategies to reduce the cost of dynamic dispatch. They analyze the code to prove that a given call site always uses a receiver of the same known class, in which case they rewrite the call as a static dispatch. For calls where the receiver's class either varies or is unknown, the implementation can cache search results to speed up subsequent calls. In such a scheme, the dispatch search looks in a method cache before it searches the class hierarchy. If the search finds the receiver's class and the method name in the cache, the call uses the cached method pointer.

SECTION REVIEW

Algol-like languages typically use lexical scoping, in which name spaces are properly nested and new instances of a name obscure older ones. Each call creates an activation record for the callee, which includes the callee's local data area. Object-oriented languages add an inheritance-based hierarchy to the name space, based on the class definitions. This dual-hierarchy name space leads to more complex interactions among names and to more complex implementations.

Both styles of naming require runtime structures to reflect and implement the naming hierarchy. In an ALL, activation records capture the structure of the name space, provide the necessary storage for most values, and preserve the state necessary for correct execution. In an OOL, activation records for methods still capture the lexically scoped part of the name space and the execution state. However, the implementation also needs a hierarchy of object records to capture the inheritance-based name space.

REVIEW QUESTIONS

1. In C, `setjmp` and `longjmp` provide a mechanism for interprocedural transfer of control. `setjmp` builds a structure to encapsulate the runtime environment; invoking `longjmp` on that environment restores the environment and lets execution continue as if the most recent `setjmp` had just returned. What information must `setjmp` preserve? How does the implementation of `setjmp` change between stack-allocated and heap-allocated ARs?
2. Consider the example from Fig. 6.3. If the programmer casts `aColorPoint` to class `Point`, what actions must the generated code take to enforce that cast and to produce the correct behavior? (Recall that the effect of a cast is local, so that the code should not permanently change the receiver's OR.

6.4 PASSING VALUES BETWEEN PROCEDURES

The central notion underlying the concept of a procedure is abstraction. The programmer abstracts common operations relative to a small set of names, or formal parameters, and encapsulates those operations in a procedure. To use the procedure, the programmer invokes it with an appropriate *binding* of values, or actual parameters, to those formal parameters. The callee executes, using the formal parameter names to access the values passed as actual parameters. If the programmer desires, the procedure can return a result.

6.4.1 **Passing Parameters**

Parameter binding maps the actual parameters at a call site to the callee's formal parameters. It lets the programmer write a procedure without knowledge of the contexts in which it may be called. It lets the programmer invoke the procedure from many distinct contexts without exposing details of the procedure's internal operation to each of its callers. Thus, parameter binding plays a critical role in our ability to write abstract, modular code.

Most modern programming languages use one of two conventions for mapping actual parameters to formal parameters: *call-by-value* binding and *call-by-reference* binding. These techniques differ in their behavior. The distinction between them may be best explained by understanding their implementations.

Call by Value

Call by value
a convention where the caller evaluates the actual parameters and passes their values to the callee

Any modification to a call-by-value parameter in the callee is not visible in the caller.

Consider the following C code—a procedure fee and several call sites that invoke fee:

```
int fee(int x, int y) {           c = fee(2,3);
    x = 2 * x;                    a = 2;
    y = x + y;                    b = 3;
    return y;                     c = fee(a,b);
}                                 a = 2;
                                  b = 3;
                                  c = fee(a,a);
```

With *call-by-value* parameter passing, as in C, the caller copies the value of an actual parameter into the appropriate location for the corresponding formal parameter—either a register or a parameter slot in the callee's AR. Only one name refers to that value—the name of the formal parameter. Its value is an initial condition, determined by evaluating the actual parameter at the time of the call. If the callee changes its value, that change is visible inside the callee, but not in the caller.

The three invocations produce the following results when invoked using call-by-value parameter binding:

Call by	a		b		Return
Value	in	out	in	out	Value
fee(2,3)	-	-	-	-	7
fee(a,b)	2	2	3	3	7
fee(a,a)	2	2	3	3	6

With call-by-value, the binding is simple and intuitive.

One variation on call-by-value binding is *call-by-value-result* binding. In this scheme, the values of formal parameters are copied back into the corresponding actual parameters as part of the return from the callee. The programming language ADA includes value-result parameters. Some FORTRAN 77 implementations used value-result binding.

Call by Reference

With a *call-by-reference* parameter, the caller passes an address rather than a value. If the actual parameter resides in memory, the caller passes its memory address. If the actual parameter is an expression, the caller evaluates the expression, stores its value into the caller's local data area, and passes the address of that location. Values kept in registers and constants should be handled in the same way as expressions. Inside the callee, each reference to a formal parameter needs an extra level of indirection.

Call by reference
a convention where the compiler passes an address for the actual parameter to the callee

If the actual parameter is a variable, then changing the formal's value also changes the actual's value.

Call by reference differs from call by value in two critical ways. First, if the caller passes a variable *x* as a call-by-reference actual parameter bound to *y* in the callee, then any change to *y* is also a change to *x*. Second, if the callee can access *x* directly, then it has two names inside the callee, which can lead to counterintuitive behavior. When the callee has two names for one storage location, we say that the names are *aliases*.

Consider the earlier example, rewritten in PL/I, which uses call-by-reference parameter binding.

```
fee: procedure (x,y)                 c = fee(2,3);
         returns fixed binary;       a = 2;
     declare x, y fixed binary;      b = 3;
     x = 2 * x;                      c = fee(a,b);
     y = x + y;                      a = 2;
     return y;                       b = 3;
     end fee;                        c = fee(a,a);
```

With call-by-reference parameter binding, the example produces different results. The first call is straightforward. The second call redefines both a

Alias
When two names can refer to the same location, they are said to be *aliases*.

In the example, the call fee(a,a) creates an alias between x and y inside fee.

and b; those changes would be visible in the caller. The third call causes x and y to refer to the same location, and thus, the same value. This *alias* changes fee's behavior. The first assignment sets both a and b to the value 4. The second assignment then sets both a and b to the value 8, and fee returns 8. A call to fee(2,2), with literal constants, would instead return 6.

Call by	a		b		Return
Reference	in	out	in	out	Value
fee(2,3)	-	-	-	-	7
fee(a,b)	2	4	3	7	7
fee(a,a)	2	8	3	3	8

Space for Parameters

The size of the representation for a parameter has an impact on the cost of procedure calls. Scalar values, such as variables and pointers, are stored in registers or in the parameter area of the callee's AR. With call-by-value parameters, the actual value is stored; with call-by-reference parameters, the address of the parameter is stored. In either case, the cost per parameter is small—typically a single store operation.

Large values, such as arrays, records, or structures, pose a problem for call by value. If the language requires that large values be copied, the overhead of copying them into the callee's parameter area will add significant cost to the procedure call. (In this case, the programmer may want to model call by reference and pass a pointer to the entity rather than the entity itself.) Some languages allow the implementation to pass such entities by reference. Other languages include provisions that let the programmer specify that passing a particular parameter by reference is safe and acceptable; for example, adding the const attribute to a parameter in a C function prototype assures the compiler that the parameter is not modified by a call to the function.

6.4.2 **Returning Values**

With call-by-value parameters, some linkage conventions designate the register reserved for the first parameter as the register to hold the return value.

To return a value from a function, the compiler must set aside space for the returned value. Because the return value, by definition, is used after the callee terminates, it needs storage outside the callee's AR. If the compiler writer can ensure that the return value is of small fixed size, then it can store the value either in the caller's AR or in a designated register.

All of our pictures of the AR have included a slot for a returned value. To use this slot, the caller allocates space for the returned value in its own AR

CALL-BY-NAME PARAMETER BINDING

ALGOL introduced *call-by-name* parameter binding. In call-by-name binding, a reference to a formal parameter behaves as if the actual parameter had been textually substituted in its place, with appropriate renaming. This simple rule can lead to complex behavior. Consider the following short example in ALGOL-60:

```
begin comment Simple array example;
    procedure zero(Arr, i, j, u1, u2);
        integer Arr, i, j, u1, u2;
        begin;
            for i := 1 step 1 until u1 do
                for j := 1 step 1 until u2 do
                    Arr := 0;
            end;
    integer array Work[1:100,1:200];
    integer p, q, x, y;
    x := 100;
    y := 200;
    zero(Work[p,q], p, q, x, y);
end
```

The call to `zero` assigns zero to every element of the array `Work`. To see this, rewrite `zero` with the text of the actual parameters.

While call-by-name binding was easy to define, it was difficult to implement and to understand. In general, the compiler must produce, for each formal parameter, a function that evaluates the actual parameter to return a pointer. These functions are called *thunks*. Generating competent thunks was complex; evaluating a thunk for each parameter access was expensive. In the end, these disadvantages overcame any advantages that call-by-name parameter binding offered.

The programming language R uses call-by-name to create lazy parameters. A call creates and passes thunks, or *promises*. The code evaluates the thunk when the parameter is first referenced and stores its result for later references.

and stores a pointer to that space in the return-value slot of its own AR. The callee can load the pointer from the caller's return-value slot (using the copy of the caller's ARP that it has in the callee's AR). With that pointer, the callee can access the space in the caller's AR set aside for the returned value. As long as both caller and callee agree about the size and type of the returned value, this approach works.

If the caller cannot know the size of the returned value, the callee may need to allocate space for it, presumably on the heap. In this case, the callee allocates the space, stores the returned value there, and stores the pointer in the return-value slot of the caller's AR. On return, the caller can access the return value using the pointer that it finds in its return-value slot. The heap space must be reclaimed, explicitly by the caller or implicitly through collection.

If the return value is small—the size of the return-value slot or less— then the compiler can eliminate the indirection. Instead, the callee can store the value directly into the return value slot of the caller's AR. The caller can then use the value directly from its AR. This improvement requires, of course, that the compiler handle the value in the same way in both the caller and the callee. Fortunately, type signatures for procedures can ensure that both compiles have the requisite information.

6.4.3 Establishing Addressability for Nonlocal Variables

Base address
The address of the start of a data area is often called a *base address*.

As part of the linkage convention, the compiler must ensure that each procedure can generate an address for each variable that it needs to reference. In general, the address calculation consists of two portions: finding the *base address* of the appropriate data area for the scope that contains the value, and finding the correct offset within that data area. The problem of finding base addresses divides into three cases: data areas with static base addresses, data areas whose addresses cannot be known until runtime, and heap-allocated objects.

Variables with Static Base Addresses

Compilers typically arrange for global data areas and static data areas to have static base addresses. To generate an address for a variable in one of these areas, the compiler emits code to compute the data area's base address into a register and to add the variable's offset to that base address. These computations are so common that most ISAs include address modes to represent them and to ensure that they execute efficiently. For example, ILOC has a "register + immediate" mode (`loadAI`) and a "register + register" mode (`loadAO`).

To generate the runtime address of a static base address, the compiler attaches a symbolic, assembly-level label to the data area. Depending on the target machine's instruction set, that label might be used in a load immediate operation or it might be used to initialize a slot in the constant pool, in which case it can be moved into a register with a standard load operation.

The compiler constructs the label for a base address by *mangling* the name. Typically, it adds a prefix, a suffix, or both to the original name, using characters that are legal in the assembly code but not in the source language. For example, mangling the global variable name `fee` might produce the label `&fee.`; the label is then attached to an assembly-language pseudooperation that reserves space for `fee`. To move the address into a register, the compiler might emit an operation such as `loadI &fee.` ⇒ r_i. Subsequent operations can then use r_i to access the memory location for `fee`. The label becomes a relocatable symbol for the assembler and the loader, which convert it into a runtime virtual address.

Name mangling
the process of constructing a unique string from a source-language name

If the value of `&fee.` is too long for an immediate load operation, the compiler may need to store the address in a known location and load it from there.

Global variables may be labeled individually or in larger groups. In FORTRAN, for example, the language collects global variables into common blocks. A typical FORTRAN compiler establishes one label for each common block. It assigns an offset to each variable in each common block and generates `load` and `store` operations relative to the common block's label. If the data area is larger than the offset allowed in a "register + offset" operation, it may be advantageous to have different labels for different parts of the data area.

Similarly, the compiler may combine all the static variables in a single scope into one data area. This reduces the likelihood of an unexpected naming conflict; such conflicts are discovered during linking or loading and can be confusing to the programmer. To avoid such conflicts, the compiler can base the label on a globally visible name associated with the scope. This strategy decreases the number of base addresses in use at any time, which may reduce demand for registers. Using too many registers to hold base addresses may adversely affect overall runtime performance.

Local Variables of the Current Procedure

Accessing a local variable of the current procedure is straightforward. While its virtual address is unknowable at compile time, its base address is simply the address of the current AR, recorded in the ARP. Further, the compiler knows the variable's offset in the local data area of that AR. (The compiler assigned that offset during storage assignment and carried it in the variable's static coordinate.) Thus, the compiler can emit code that adds the variable's offset to the ARP to produce its virtual address. Almost all processors provide an address mode to support this kind of base + offset memory operation. In ILOC those operations are `loadAI`, `loadAO`, `storeAI`, and `storeAO`.

In some cases, a value is not stored at a constant offset from the ARP. The value might reside in a register, in which case loads and stores are not needed. If the variable has an unpredictable or changing size, the compiler will store it in an area reserved for variable-size objects, either at the end of the local data area or in the heap. In this case, the compiler can reserve space in the AR for a pointer to the variable's actual location, access the pointer through an ARP + offset operation, and then use the pointer to access the variable.

Local Variables of Other Procedures

To access a local variable of some enclosing lexical scope, the compiler must arrange for the construction of runtime data structures that support the task of finding the local data areas of such scopes. It must also emit code that uses those structures to find the needed virtual address.

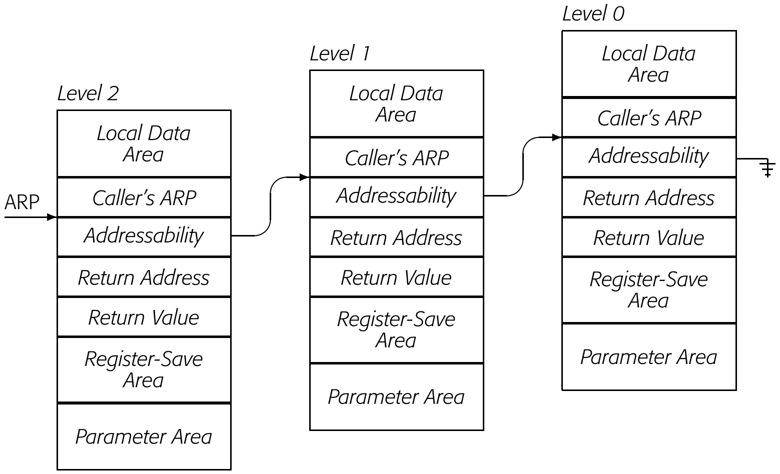
For example, assume that procedure fee, at lexical level m, references variable x defined at lexical level n in procedure fie, $n < m$. The parser can construct the static coordinate for x, $\langle n, o\rangle$, where o is x's offset in the AR for fie. To convert $\langle n, o\rangle$ into a runtime address, the compiler emits code that uses n to find the ARP of the most-recently created level n procedure, and then adds o to that ARP to compute x's runtime virtual address.

Of course, this plan relies on the availability of runtime data structures that can find the level n ARP. The compiler must emit code to build and maintain those structures. The two most common methods are called access links and a global display. The following subsections explore each of them.

Access Links

The intuition behind access links is simple. The compiler ensures that each AR contains a pointer, called an *access link* or a *static link*, to the AR of its immediate surrounding scope. The access links form a chain that includes all the lexical ancestors of the current procedure, as shown in Fig. 6.4. Thus, any local variable of another procedure that is visible to the current procedure is stored in an AR on the chain of access links that begins in the current procedure.

To access a value $\langle n,o\rangle$ from a level m procedure, the compiler emits code to walk the chain of links and find the level n ARP. Next, it emits a load that uses the level n ARP and o. To make this concrete, consider the program represented by Fig. 6.4. Assume that m is 2 and that the access link is stored at an offset of -4 from the ARP. The following table shows a set of three different static coordinates alongside the ILOC code that a compiler might generate for them. Each sequence leaves the result in r_2.

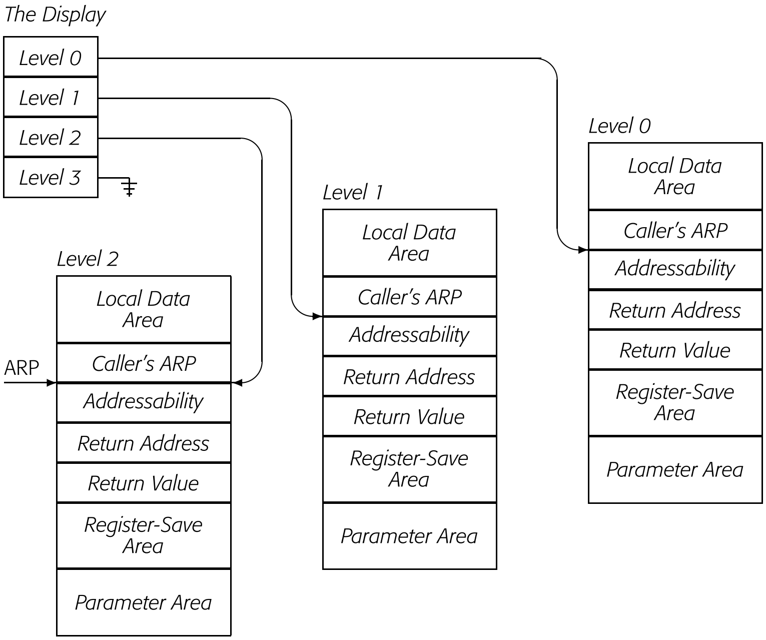


■ **FIGURE 6.4** Using Access Links.

Coordinate	Code
⟨2,24⟩	loadAI r_{arp}, 24 ⇒ r_2
⟨1,12⟩	loadAI r_{arp}, -4 ⇒ r_1
	loadAI r_1, 12 ⇒ r_2
⟨0,16⟩	loadAI r_{arp}, -4 ⇒ r_1
	loadAI r_1, -4 ⇒ r_1
	loadAI r_1, 16 ⇒ r_2

Since the compiler has the static coordinate for each reference, it can compute the static distance ($m-n$), which tells it how many chain-following loads to generate. The cost of the address calculation is proportional to the static distance. If programs exhibit shallow lexical nesting, the difference in cost between accessing two variables at different levels will be fairly small.

To maintain access links, the compiler must add code to each procedure call that finds the appropriate ARP and stores it as the callee's access link. For a caller at level p and a callee at level q, three cases arise. If $q = p + 1$, the callee is nested inside the caller, and the callee can use the caller's ARP as its access link. If $q = p$, the callee's access link is the same as the caller's access link. Finally, if $q < p$, the callee's access link is the level $q - 1$ access link for the caller. (If q is zero, the access link is null.) The compiler can generate a sequence of $p - q + 1$ loads to find this ARP and store that pointer as the callee's access link.



■ **FIGURE 6.5** Using a Global Display.

Global Display

In this scheme, the compiler allocates a single global array, called the *display*, to hold the ARP of the most recent activation of a procedure at each lexical level. References to local variables of other procedures become indirect references through the display. To access a variable with static coordinate $\langle n,o \rangle$, the compiler first retrieves the level n ARP from the display. Then, it uses that ARP along with the offset o in a base + offset load or store operation. References to local variables of the current procedure can still use the ARP directly from r_{arp}.

Fig. 6.5 shows the program state from Fig. 6.4 represented with a global display instead of access links. The display has one entry per lexical level in the program. The display, itself, can be statically allocated and assigned a known mangled label. Unused display entries contain an invalid pointer.

Using the same static coordinates as in the discussion of access links, the compiler might emit the following code for a display-based implementation. Assume that the current procedure is at lexical level 2, and that the label `_&disp_:` gives the display's address.

Coordinate	Code
⟨2,24⟩	loadAI r_{arp}, 24 ⇒ r_2
⟨1,12⟩	loadI _&disp_ ⇒ r_1
	loadAI r_1, 4 ⇒ r_1
	loadAI r_1, 12 ⇒ r_2
⟨0,16⟩	loadI _&disp_ ⇒ r_1
	loadAI r_1, 0 ⇒ r_1
	loadAI r_1, 16 ⇒ r_2

With a display, the cost of nonlocal access is fixed. With access links, the compiler generates a series of $m - n$ loads; with a display, it uses $n \times l$ as an offset into the display, where l is the length of a pointer (4 in the example). Local access is still cheaper than nonlocal access, but with a display, the penalty for nonlocal access is constant.

Of course, the compiler must insert code where needed to maintain the display. Thus, when procedure *p* at level *n* calls some procedure *q* at level *n*+1, *p*'s ARP becomes the display entry for level *n*. (While *p* is executing, that entry is unused.) The simplest way to maintain the display is to have *p* update the level *n* entry when control enters *p*, to replace that entry with its own ARP, and to restore the original entry on exit from *p*.

p can save the old level *n* display entry in the addressability slot of *p*'s own AR.

The compiler can avoid some of these display updates. If procedure *p* does not call procedures that are nested within it, then its own ARP cannot be used in a nonlocal access and *p* need not update the display. This observation eliminates many of the unneeded updates.

To do a complete job of eliminating unneeded updates, the compiler would need to know which nonlocal variables each procedure referenced.

SECTION REVIEW

The ability to map values into and out of procedures is part of what makes procedures useful. Two distinct mechanisms are involved: parameter binding and direct access to names in surrounding scopes. For parameters, languages specify different mechanisms that the compiler must implement. For nonlocal accesses, the compiler must plan and maintain runtime structures that point to the appropriate data areas.

The most confusing aspect of this material is the distinction between compile-time actions, such as the parser computing a static distance coordinate, and runtime actions, such as the code tracing up a chain of access links. The compiler directly performs the compile-time actions. For the runtime actions, it emits code to perform those actions at runtime.

```
subroutine change(n)
  integer n
  n = n * 2
end

program test
  call change(2)
  print *, 2 * 2
end
```

REVIEW QUESTIONS

1. An early FORTRAN implementation had an odd bug. The short program in the margin would print, as its result, the value 16. What did the compiler do that led to this result? What should it have done instead? (FORTRAN uses call-by-reference parameter binding.)
2. One issue in the implementation of closures is the need to keep the relevant ARs alive once a closure is created. What role can access links or a display play in this process? Is one technique better than the other in this situation?

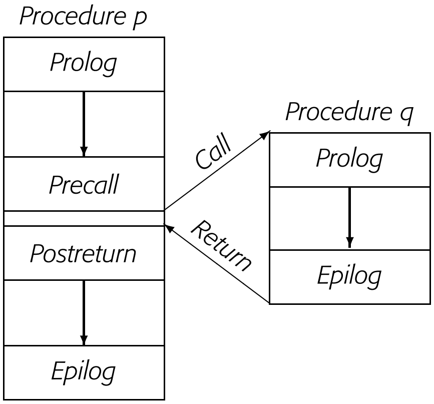
6.5 STANDARDIZED LINKAGES

The procedure linkage is a contract between the compiler, the operating system, and the target machine that clearly divides responsibility for naming, allocation of resources, addressability, and protection. The procedure linkage ensures interoperability of procedures between the user's code, as translated by the compiler, and code from other sources, including system libraries, application libraries, and code written in other programming languages. Typically, all of the compilers for a given combination of target machine and operating system use the same linkage, to the extent possible.

The linkage convention isolates each procedure from the different environments found at call sites that invoke it. Assume that procedure p has an integer parameter x. Different calls to p might bind x to a local variable stored in the caller's stack frame, to a global variable, to an element of some static array, or to the result of evaluating an integer expression such as $y+2$. Because the linkage convention specifies how to evaluate the actual parameter and store its value, as well as how to access x in the callee, the compiler can generate code for the callee that ignores the differences between the runtime environments at the different calls sites. As long as all the procedures obey the linkage convention, the details will mesh to create the seamless transfer of values promised by the source-language specification.

The linkage convention is, of necessity, machine dependent. For example, it depends implicitly on information such as the number of registers available on the target machine and the mechanisms for executing a call and a return.

Fig. 6.6 shows how the pieces of a standard procedure linkage fit together. Each procedure has both a *prolog* and an *epilog sequence*. Each call site includes both a *precall* and a *postreturn sequence*.



■ **FIGURE 6.6** A Standard Procedure Linkage.

Precall Sequence

The precall sequence begins the process that both preserves the caller's environment and creates the callee's environment. It allocates space for the callee's AR. It evaluates the actual parameters and stores the values or addresses in the designated slot in the callee's AR. It stores the address of the postreturn sequence as the return address. It stores the caller's ARP in the callee's AR. It saves any caller-saves registers that must be preserved into the register-save area of the caller's AR. It may also perform maintenance on the addressability data structures: the access links or the display.

If a call-by-reference parameter resides in a register at the call site, the precall may need to store it into the caller's AR to create an address to pass to the callee.

To activate the callee, the precall sequence writes the callee's ARP into the register designated to hold the current ARP and jumps to the callee's prolog code.

Prolog Sequence

The prolog sequence finishes the process that both preserves the caller's environment and creates the callee's new environment. It saves any callee-saves registers that the callee will use. It may need to allocate space for the callee's local data area. It may need to load the base addresses for any static or global data areas that the callee references. It may also perform maintenance on the addressability data structures: the access links or the display. Finally, it performs any initializations of local variables. At that point, it begins to execute the code for the callee's body.

Epilog Sequence

The epilog sequence starts the process that dismantles the callee's environment and reestablishes the caller's environment. If the procedure returns a value, the epilog may store that value into the address specified by the caller. (Alternatively, the code generated for a return statement may perform this task.) The epilog restores any callee-saves registers that the prolog saved. It may also perform maintenance on the addressability data structures: the access links or the display. If the callee's local data area was allocated separately from its AR, the epilog frees that space. Finally, it loads the return address, restores the caller's ARP, and jumps to the return address.

Postreturn Sequence

The postreturn sequence completes the process that restores the caller's environment. It frees the callee's AR. It restores any call-by-reference actual parameters that need to be returned to registers. It restores any caller-saves registers that the precall sequence saved. Finally, it continues execution of the caller at the point after the call.

This framework provides general guidance for building a linkage convention. Many of the tasks can be shifted between caller and callee. In general, moving work into the prolog and epilog code produces more compact code. The precall and postreturn sequences are generated for each call, while the prolog and epilog occur once per procedure. If procedures are called, on average, more than once, then there are fewer prolog and epilog sequences than precall and postreturn sequences.

Many of the values shown in the AR diagrams can be passed to the callee in registers. The return address, the address for the return value, and the caller's ARP are obvious candidates. Many conventions pass the first *k* actual parameters in registers—a typical value for *k* might be 3 or 4. If the call has more than *k* parameters, the remaining actual parameters can be stored in the callee's AR.

Saving Registers

A value is *live* across a call if its value is created before the call and that value is used after the call.

At some point in the call sequence, any register values that the caller expects to survive across the call must be saved into memory. Either the caller or the callee can perform the actual save; each choice has advantages and disadvantages. If the caller saves the registers, it can avoid saving values that it knows are not live across the call; that knowledge might allow it to save and restore fewer values. Similarly, if the callee saves the registers, it can avoid saving values in registers that it does not use; again, knowledge of register use in the callee might reduce saves and restores at the call.

In general, the compiler can use its knowledge of the procedure being compiled to optimize register save behavior. For any specific division of labor between caller and callee, we can construct programs for which it works

MORE ABOUT TIME

In a typical system, the linkage convention is negotiated between compiler writers and operating-system implementors at an early stage of system development. Thus, issues such as the distinction between caller-saves and callee-saves registers are decided at design time. When the compiler runs, it must emit procedure prolog and epilog sequences for each procedure, along with precall and postreturn sequences for each call site. This code executes at runtime. The compiler cannot know the return address that it should store into a callee's AR. (Neither can it know, in general, the address of that AR.) It can, however, include a mechanism that will generate the return address at link time (using a relocatable assembly language label) or at runtime (using some offset from the program counter) and store it into the appropriate location in the callee's AR.

Similarly, in a system that uses a display to provide addressability for local variables of other procedures, the compiler cannot know the runtime addresses of the display or the AR. Nonetheless, it emits code to maintain the display. The mechanism that achieves this requires two pieces of information: the lexical nesting level of the current procedure and the address of the global display. The former is known at compile time; the latter can be determined at link time by using a relocatable assembly language label. Thus, the prolog can simply load the current display entry for the procedure's level (using a `loadAO` from the display address) and store it into the AR (using a `storeAO` relative to the ARP). Finally, it stores the address of the new AR into the display slot for the procedure's lexical level.

well and programs for which it does not. Most modern systems take a middle ground and designate a portion of the register set for caller-saves treatment and a portion for callee-saves treatment. In practice, this seems to work well. It encourages the compiler to put long-lived values in callee-saves registers, where they will be stored only if the callee actually needs the register. It encourages the compiler to put short-lived values in caller-saves registers, where it may avoid saving them at a call.

Allocating the Activation Record

In the general case, both the caller and the callee need access to the callee's AR. Unfortunately, the caller cannot know, in general, how large the callee's AR must be (unless the compiler and linker can contrive to have the linker paste the appropriate values into each call site).

With stack-allocated ARs, a middle ground is possible. Since allocation consists of incrementing the stack-top pointer, the caller can begin the creation of the callee's AR by bumping the stack top and storing values into the appropriate places. When control passes to the callee, it can extend the partially built AR by incrementing the stack top to create space for local data. The postreturn sequence can then reset the stack-top pointer, performing the entire deallocation in one step.

With heap-allocated ARs, it may not be possible to extend the callee's AR incrementally. In this situation, the compiler writer has three choices.

1. The compiler can pass the values that it must store in the callee's AR in registers or in the caller's own AR. The prolog sequence can then allocate an appropriately sized AR and store the passed values in it. The parameter values might go into the caller's AR. Access to those parameters could use the copy of the caller's ARP that is saved in the callee's AR.
2. The compiler writer can split the AR into multiple distinct pieces: one to hold the parameter and control information generated by the caller and the others to hold space needed by the callee but of a size unknown to the caller. Because the caller cannot, in general, know how large to make the local data area, it can let the callee allocate its own local data area. The callee might dedicate a register to hold the base address of that local data area.
3. The compiler could use a scheme where each procedure creates an initialized static constant using a mangled label. The compiler can store procedure *fee*'s local data area size in a label mangled from "*fee*." Any call to *fee* can load that constant and use it to size the local data area.

Heap-allocated ARs increase the cost of a procedure call. Careful implementation can reduce those costs.

Managing Displays and Access Links

Whether the compiler writer chooses to use access links or a display, either mechanism requires some work in the calling sequence. With a display, the prolog updates the display record for its own level and the epilog restores it. If the procedure never calls a more deeply nested procedure, it can skip this step. With access links, the precall must find the appropriate first access link for the callee. The amount of work varies with the difference in lexical level between caller and callee. As long as the callee is known at compile time, either scheme is reasonably efficient. If the callee is unknown (if it is, for example, a function-valued parameter), the compiler may need to emit special-case code to perform the appropriate steps.

SECTION REVIEW

The linkage convention is a social contract between the compiler, the operating system, and the underlying hardware. It governs the transfer of control between procedures, the preservation of the caller's state, the creation of the callee's state, and the rules for passing values between them.

Standardized linkages enable the assembly of executable programs from code written by different people and compiled at different times. They ensure that each procedure can operate safely and correctly. The same conventions let applications invoke system calls and library routines. While the details of the linkage convention vary across systems, the basic concepts are similar across most target machines, operating systems, and compilers.

REVIEW QUESTIONS

1. What role does the linkage convention play in the construction of interlanguage programs? What facts might the compiler need to know in order to generate code for an interlanguage call?
2. If, at a call, the compiler knows that the callee does not contain any procedure calls, what steps might it omit from the calling sequence? Are there fields in the AR that the callee would never need?

6.6 ADVANCED TOPICS

The compiler must arrange for the allocation of space to hold both user defined data and runtime support structures. Often, those structures have lifetimes that do not fit well into the first-in first-out discipline of a stack. In such cases, the language implementation allocates space in the runtime heap—a region of memory set aside for such objects and managed by routines in a runtime support library. The compiler must also arrange storage for other objects that have lifetimes unrelated to the flow of control, such as most of the objects in a JAVA program.

We assume a simple interface to the heap, namely, a routine `allocate(size)` and a routine `free(address)`. The `allocate` routine takes an integer argument `size` and returns the address of a block of space in the heap that contains at least `size` bytes. The `free` routine takes the address of a block of previously allocated space in the heap and returns it to the pool of free space. The critical issues that arise in designing algorithms for explicitly managing the heap are the efficiency of both `allocate` and `free` and the extent to which the pool of free space becomes fragmented into small blocks.

This section sketches the algorithms involved in allocation and reclamation of space in a runtime heap. Section 6.6.1 describes first-fit allocation for the heap and expands on that technique to develop the notion of multipool allocators. Section 6.6.2 examines implicit schemes for storage deallocation and reclamation, which eliminate the need for the code to explicitly `free` heap-based storage.

6.6.1 Explicit Heap Management

Most language implementations include a runtime system that provides support functions for the code generated by the compiler. That support typically includes routines to manage a runtime heap. The heap management support may be language specific, as in a SCHEME interpreter or a JAVA virtual machine, or it may be part of the operating system, as in the POSIX implementations of `malloc` and `free`.

Many techniques have been proposed to implement `allocate` and `free`. Most of those implementations share common strategies and insights. This section explores a simple strategy, *first-fit allocation*, that exposes most of the issues, and then shows how a strategy such as first fit is used to implement a modern allocator.

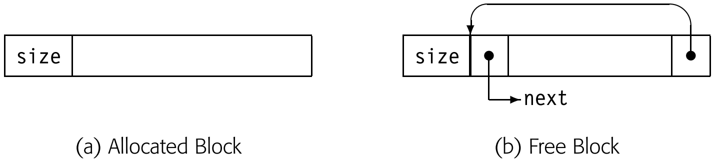
First-Fit Allocation

The primary focus of a first-fit allocator is the speed of `allocate` and `free`. Speed takes precedence over efficient use of memory. Blocks contain a hidden size field, typically located before the address returned by allocate; see Fig. 6.7(a). The allocator maintains a list of available blocks, called the *free list*. Blocks on the free list have additional fields, as shown in Fig. 6.7(b). Each free block keeps a pointer to the next block on the free list (set to null in the last block) and a pointer to the block itself in its last word of the block. The allocator initializes the heap to contain a single large block of free memory.

To reduce fragmentation, the allocator can maintain a minimum block size, s. It will then only split a block if both of the resulting blocks have size greater than or equal to s.

A call to `allocate(`k`)` causes the following sequence of events: The `allocate` routine walks the free list until it discovers a block with size greater than or equal to k plus one word for the `size` field. Assume that it finds an appropriate block, b_i. It removes b_i from the free list. If b_i is larger than necessary, `allocate` creates a new free block from the excess space at the end of b_i, updates all of the relevant fields in b_i and the new block, and puts the new block on the free list. Finally, `allocate` returns a pointer to the appropriate offset inside b_i.

If `allocate` does not find a large enough block, it tries to extend the heap. If it succeeds, it returns a block of appropriate size from this newly allocated portion of the heap. If extending the heap fails, `allocate` reports failure.



■ **FIGURE 6.7** Blocks in a First-Fit Allocator.

To deallocate a block, the program calls `free` with the address of the block, b_j. The simplest implementation of `free` adds b_j to the head of the free list and returns. This approach is fast. Unfortunately, it leads to an allocator that, over time, fragments memory into small blocks.

To overcome this flaw, the allocator can use the pointer at the end of a freed block to coalesce adjacent free blocks. The `free` routine loads the word preceding b_j's size field, which is the end-of-block pointer for the block that immediately precedes b_j in memory. If that word contains a valid pointer, and it points to a matching block header (one whose address plus size field points to the start of b_j), then both b_j and its predecessor are free. The `free` routine can combine them by increasing the predecessor's size field and storing the appropriate pointer at the end of $\mathtt{b}_\mathtt{j}$. If the predecessor block is allocated, then `free` adds b_j to the start of the free list.

To make this scheme work, `allocate` and `free` must maintain the end-of-block pointers. Each time that `free` processes a block, it must update that pointer with the address of the head of the block. The `allocate` routine must invalidate either the `next` pointer or the end-of-block pointer to prevent `free` from coalescing a freed block with an allocated block in which those fields have not been overwritten.

The `free` routine can also try to combine $\mathtt{b}_\mathtt{j}$ with its successor in memory, $\mathtt{b}_\mathtt{k}$. It can use $\mathtt{b}_\mathtt{j}$'s size field to locate the start of $\mathtt{b}_\mathtt{k}$. It can use $\mathtt{b}_\mathtt{k}$'s size field and end-of-block pointer to determine if $\mathtt{b}_\mathtt{k}$ is free. If $\mathtt{b}_\mathtt{k}$ is free, then `free` can combine the two blocks by removing $\mathtt{b}_\mathtt{k}$ from the free list, adding $\mathtt{b}_\mathtt{j}$ to the free list, and updating $\mathtt{b}_\mathtt{j}$'s size field and end-of-block pointer appropriately. To make the free-list update efficient, the free list should be a doubly-linked list. The overhead for the doubly-linked list is minimal.

As described, the coalescing scheme depends on the fact that the relationship between the pointers in a free block will not occur in an allocated block. While it is extremely unlikely that the allocator will identify an allocated block as free, it could happen. To ensure against this event, the implementor can require that the end-of-block pointer exists in both allocated and free blocks. If `allocate` sets the pointer to zero and `free` sets it to the block's own address, the allocator should never misidentify an allocated block.

ARENA-BASED ALLOCATION

Inside the compiler itself, the compiler writer may find it profitable to use a specialized allocator. Compilers have phase-oriented activity, which lends itself well to an arena-based allocation scheme.

With an arena-based allocator, the program creates an arena—a pool of contiguous memory in the heap—at the beginning of an activity. It uses the arena to hold allocated objects that are related in their use. Calls to allocate objects in the arena are satisfied in a stack-like fashion. An allocation involves incrementing a pointer to the arena's high-water mark and returning a pointer to the newly allocated block. No call is used to deallocate individual objects; they are freed en masse when the arena that contains them is deallocated.

The arena-based allocator is a compromise between traditional allocators and collecting allocators. With an arena-based allocator, the calls to `allocate` can be made lightweight (as in a multipool allocator). No freeing calls are needed; the program frees the entire arena in a single call when it finishes the activity for which the arena was created.

Many variations on first-fit allocation have been tried. They trade off the cost of `allocate`, the cost of `free`, the amount of fragmentation produced by a long series of allocations, and the amount of space wasted by returning blocks larger than requested.

Multipool Allocators

Multipool allocators build on first-fit allocation but simplify it based on observations about the behavior of programs. As memory sizes grew in the early 1980s, it became reasonable to waste some space if doing so led to faster allocation. At the same time, studies of program behavior suggested that real programs allocate memory frequently in a few common sizes and infrequently in large or unusual sizes.

Modern allocators use separate memory pools for several common sizes. Typically, selected sizes are powers of two, starting with a small block size (such as 32 or 64 bytes) and running up to the size of a virtual-memory page (typically 4096 or 8192 bytes). Each pool has only one size of block, so `allocate` can return the first block on the appropriate free list, and `free` can simply add the block to the head of the appropriate free list. For requests larger than a page, a separate first-fit allocator is used. Allocators based on these ideas are fast. They work particularly well for heap allocation of ARs.

These changes simplify both `allocate` and `free`. The `allocate` routine must check for an empty free list and add a new page of appropriately sized blocks to the free list if it is empty. The `free` routine inserts the freed block at the head of the free list for its size. A careful implementation could determine the size of a freed block by checking its address against the memory segments allocated for each pool. Alternative schemes include using a size field as before, and, if the allocator places all the storage on a page into a single pool, storing the size of the blocks in a page in a designated word on the page.

Debugging Help

Programs written with explicit allocation and deallocation are notoriously difficult to debug. It appears that programmers have difficulty deciding when to free heap-allocated objects. If the allocator can quickly distinguish between an allocated object and a free object, then the heap-management software can provide the programmer with some help in debugging.

Decades of experience suggest that programmers are not effective at freeing all the storage that they allocate. They also tend to free objects to which they retain pointers. Implicit deallocation fixes both problems.

For example, to coalesce adjacent free blocks, the allocator needs a pointer from the end of a block back to its head. If an allocated block has that pointer set to an invalid value, then the deallocation routine can check that field and report a runtime error when the program attempts to deallocate a free block or an illegal address—a pointer to anything other than the start of an allocated block.

For a modest additional overhead, heap-management software can provide additional help. By linking together allocated blocks, the allocator can create an environment for memory-allocation debugging tools. A snapshot tool can walk the list of allocated blocks. Tagging blocks by the call site that created them lets the tool expose memory leaks. Timestamping them allows the tool to provide the programmer with detailed information about memory use. Tools of this sort can provide invaluable help in locating blocks that are never deallocated.

6.6.2 **Implicit Deallocation**

Many programming languages support implicit deallocation of heap objects. The runtime system deallocates memory objects automatically when they are no longer in use. To perform implicit deallocation, or *garbage collection*, the compiler and runtime system must include a mechanism for determining when an object is no longer of interest, or *dead*, and a mechanism for reclaiming and recycling the dead space.

Garbage collection
the implicit deallocation of objects on the runtime heap

Dead
An object is *dead* when the running code can no longer reach it.

The work associated with storage reclamation can be performed incrementally for individual statements, or it can be performed as a batch-oriented task that runs on demand when the free-space pool is exhausted. *Reference counting* is a classic technique for incremental reclamation. *Mark-sweep collection* is a classic approach to performing batch-oriented reclamation.

Reference Counting

This technique adds a counter to each heap-allocated object. The counter tracks the number of outstanding pointers, or references, that refer to the object. When the allocator creates the object, it sets the reference count to one. Each assignment to a pointer variable adjusts two reference counts. It decrements the reference count of the pointer's preassignment object and increments the reference count of the pointer's postassignment object. When an object's reference count drops to zero, no pointer exists that can reach the object, so the system may safely free the object.

Freeing an object can, in turn, discard pointers to other objects. The code must then decrement the reference counts of those objects. For example, discarding the last pointer to a tree's root should free the entire tree. When the root node's reference count drops to zero, it is freed and its descendants' reference counts are decremented. This, in turn, should free the descendants, decrementing the counts of their children. This process continues until the entire tree has been freed.

The presence of pointers in allocated objects creates several challenges for reference-counting schemes, as follows:

1. The code needs a way to distinguish a pointer from other data in the object. The system can keep a header field on each object where it stores extra information, such as a bitmap that identifies which words in the object are pointers. It can limit the range of pointers to less than a full word and use the remaining high-order bits to "tag" the pointer. It can keep detailed type information at runtime and use that to identify pointers. Note that batch collectors face this same issue and use the same solutions.
2. A decrement to one reference count, such as the root of a tree, can create a large amount of work. If external constraints require bounded deallocation times, as in real-time systems, the runtime system can adopt a scheme that limits the number of objects freed on each pointer assignment. It can keep a queue of objects to free and limit the number freed on each reference-count adjustment. This approach distributes the cost over a larger set of operations and bounds the work done per assignment.
3. The program might build a cyclic structure in the heap. The reference counts in a cyclic data structure will not reach zero. When the last

external pointer is discarded, the cycle becomes unreachable; its storage cannot be freed by the reference count mechanism. To ensure that cyclic structures are freed, the *programmer* must break the cycle before discarding the last pointer to the cycle. Many heap-allocated objects, such as variable-length strings and ARs, cannot be involved in such cycles.

Reference counting incurs additional work on every pointer assignment. The amount of work done for a specific pointer assignment can be bounded; in any well-designed scheme, the total cost can be limited to some constant factor times the number of pointer assignments executed plus the number of objects allocated. Proponents of reference counting argue that these overheads are small enough and that the pattern of reuse in reference-counting systems produces good program locality. Opponents of reference counting argue that real programs do more pointer assignments than allocations, so that garbage collection achieves equivalent functionality with less total work.

Batch Collectors

Batch collectors consider deallocation only when the free-space pool has been exhausted. When the allocator fails to find needed space, it invokes the batch collector. The collector pauses the program's execution, examines the pool of allocated memory to discover unused objects, and reclaims their space. When the collector terminates, the free-space pool should be nonempty. The allocator can finish its original task and return a newly allocated object to the caller. (As with reference counting, schemes exist that perform collection incrementally to amortize the cost over longer periods of execution.)

If the collector cannot free any space, then it must request additional space from the system. If none is available, allocation fails.

Logically, batch collectors proceed in two phases. The first phase discovers the set of objects that can be reached from pointers stored in program variables and compiler-generated temporaries. The collector conservatively assumes that any object reachable in this manner is live and that the remainder is dead. The second phase deallocates and recycles dead objects. Two commonly used techniques are *mark-sweep* collectors and *copying* collectors. They differ in their implementation of the second phase of collection—recycling.

Identifying Live Data

Collecting allocators discover live objects with a marking algorithm. The collector needs a bit for each object in the heap, called a *mark bit*. Conceptually, think of the mark bits as stored in the objects. Mark bits should be created in the "unmarked" or "clear" state.

```
Clear all marks
Worklist ← { pointer values from activation records & registers }

while (Worklist ≠ ∅) do
    remove p from the Worklist
    if (p→object is unmarked) then
        mark p→object
        add pointers from p→object to Worklist
```

■ **FIGURE 6.8** A Simple Marking Algorithm.

The initial step of the sweep phase builds a worklist that contains all the pointers stored in registers and in variables accessible to active procedures. The second phase of the algorithm walks forward from these pointers and marks every object that is reachable from this set of visible pointers.

Fig. 6.8 shows a high-level sketch of a simple marking algorithm. It halts because the heap is finite and the marks ensure that each pointer enters the *Worklist* at most once. The cost of marking is proportional to the number of pointers that it examines.

The marking algorithm can be either precise or conservative. The difference lies in how the algorithm determines that a specific data value is a pointer in the final line of the *while* loop.

- In a precise marking phase, the compiler and runtime system know the type and layout of each object. This information can be recorded in object headers, or it can be known implicitly from the type system. Either way, the marking phase only follows real pointers.
- In a conservative marking phase, the compiler and runtime system may be unsure about the type and layout of some objects. Thus, when an object is marked, the system considers each field that may be a possible pointer. If its value might be a pointer, it is treated as a pointer. Any value that does not represent a word-aligned address might be excluded, as might values that fall outside the known boundaries of the heap.

Conservative collectors have limitations. They fail to reclaim some objects that a precise collector would find. Nonetheless, conservative collectors have been successfully retrofitted into implementations for languages such as C that do not normally support garbage collection.

When the marking algorithm halts, any unmarked object must be unreachable from the program. Thus, the second phase of the collector can treat that object as dead. Some objects marked as live may also be dead. However, the collector lets them survive because it cannot prove them to be dead. As the second phase traverses the heap to collect unused storage, it can clear the mark bits.

Mark-Sweep Collectors

Mark-sweep collectors reclaim and recycle objects in a linear pass over the heap. The collector adds each unmarked object to the free list (or one of the free lists) where the allocator will find it and reuse it. With a single free list, the same collection of techniques used to coalesce blocks in the first-fit allocator applies. To compact the heap, the collector can incrementally shuffle live objects downward during the sweep or with a postsweep compaction pass.

Copying Collectors

Copying collectors divide memory into two pools, an *old* pool and a *new* pool. The allocator always operates from the old pool. The simplest copying collectors are called *stop and copy* collectors. When an allocation fails, a stop and copy collector copies all the live data from the old pool into the new pool and swaps the identities of the old and new pools. The act of copying live data compacts it; after collection, all the free space is in a single contiguous block.

Copying can be done in a separate pass, as in mark sweep, or it can be performed incrementally, as live data is identified. An incremental scheme can mark objects in the old pool as it copies them to avoid copying the same object multiple times.

Generational collectors are an important family of copying collectors. They capitalize on the observation that once an object survives one collection, it is more likely to survive subsequent collections. A generational collector uses two "new" pools that it maintains by copying; these pools are sometimes called *nurseries*. A generational collector also maintains one or more older or *stable* pools. When the nursery becomes too crowded, the allocator collects and copies the live data from the nursery into the stable pool. Successive collections of the nursery only examine newly allocated objects. Generational schemes vary in terms of how many stable pools they maintain and how often they collect those stable pools.

Comparing the Techniques

Garbage collection frees the programmer from thinking about when to release memory and from tracking down the inevitable storage leaks that result from explicit deallocation. While the individual schemes have strengths and weaknesses, the practical benefits of implicit deallocation outweigh the disadvantages for most applications.

Reference counting distributes the cost of deallocation more evenly across program execution than does batch collection. However, it increases the cost of every assignment that involves a heap-allocated value—even if the program never runs out of free space. By contrast, batch collectors incur no cost until the allocator fails to find needed space. At that point, however, the program incurs the full cost of collection. Furthermore, any allocation can provoke a collection.

Mark-sweep collectors examine the entire heap, while copying collectors only examine the live data. Copying collectors actually move every live object, while mark-sweep collectors leave them in place. The tradeoff between these costs will vary with the application's behavior and with the actual costs of various memory references.

PYTHON reference counts objects, but invokes a batch collector when it exhausts the heap. The batch collector can reclaim "lost" cyclic structures.

Reference-counting implementations and conservative batch collectors have problems recognizing cyclic structures because they cannot distinguish between references from within the cycle and those from without. The mark-sweep collectors start from an external set of pointers, so they discover that a dead cyclic structure is unreachable. The copying collectors, starting from the same set of pointers, simply fail to copy the objects involved in the cycle.

Copying collectors compact memory as a natural part of the process. The collector can either update all the stored pointers, or it can require use of an indirection table for each object access. A precise mark-sweep collector can compact memory, too. The collector would move objects from one end of memory into free space at the other end. Again, the collector can either rewrite the existing pointers or mandate use of an indirection table.

In general, a good implementor can make both mark sweep and copying work well enough that they are acceptable for most applications. In applications that cannot tolerate unpredictable overhead, such as real-time controllers, the runtime system must incrementalize the process, as the amortized reference-counting scheme does. Such collectors are called *real-time collectors*.

6.7 SUMMARY AND PERSPECTIVE

The primary rationale for moving beyond assembly language was to provide a more abstract programming model and produce improvements in programmer productivity and program understandability. The compiler must be prepared to translate each abstraction that the source language provides.

This chapter explored techniques used to implement the abstractions introduced by procedure calls.

Procedural programming was invented early in the history of programming. Some of the first procedures were debugging routines written for early computers. These prewritten routines allowed programmers to understand the runtime state of an errant program. Without such routines, simple tasks, such as examining the contents of a variable, tracing the call stack, or printing a dump of the contents of memory, could only be achieved if the programmer entered a long sequence of machine-language code without error.

The advent of lexical scoping in languages such as ALGOL 60 influenced language design for decades. Most modern programming languages inherit some of ALGOL's philosophy on naming and addressability. Techniques invented to reduce the runtime cost of lexical scoping, such as access links and displays, are widely used today.

OOLs take the scoping concepts of ALLs and reorient them in data-directed ways. These languages typically support both a lexical scoping hierarchy and an inheritance hierarchy. The compiler for an OOL uses both compile-time and runtime structures invented for lexical scoping to model and to implement the naming discipline imposed by the inheritance hierarchy.

As new programming paradigms emerge, they will introduce new abstractions that require careful thought and implementation. By studying the successful techniques of the past and understanding the constraints and costs involved in real implementations, compiler writers will develop strategies that decrease the runtime penalty for using higher levels of abstraction.

CHAPTER NOTES

Much of the material in this chapter comes from the accumulated experience of the compiler-construction community. The best way to learn more about the name-space structures of various languages is to consult the language definitions themselves. These documents are a necessary part of a compiler writer's library.

Procedures appeared in the earliest high-level languages—that is, languages that were more abstract than assembly language. FORTRAN [28] and ALGOL 60 [282] both had procedures with most of the features found in modern languages. Early ALGOL 60 implementations developed many of the key ideas that arise in linkage conventions; for example, Ingerman first described thunks for call-by-name parameter passing in 1961 [213]. Object-oriented languages appeared in the late 1960s with SIMULA-67 [287] followed by SMALLTALK 72 [242]. The various SMALLTALK systems used method caches to reduce the cost dynamic dispatch [137,242].

ALGOL 60 introduced lexical scoping; it persists to the present day. Early ALGOL compilers introduced most of the support mechanisms described in this chapter; they implemented activation records, access links, displays, and parameter-passing techniques. Most of the ideas from Sections 6.3 through 6.5 were present in one or more of these early systems [303]. Optimizations to the linkage convention quickly appeared, such as folding storage for a block-level scope into the activation record for the procedure that contained the block. Murtagh took a systematic approach to coalescing activation records, combining the ARs of procedures that are frequently called together [281]. The IBM 370 linkage convention recognized the difference between leaf procedures and others; they avoided allocating a register save area for leaf routines.

The classic reference on memory allocation schemes is Knuth's *Art of Computer Programming* [240, § 2.5]. Modern multipool allocators appeared in the early 1980s. Reference counting dates to the early 1960s and has been used in many systems [102,136]. Schorr and Waite describe an early marking algorithm [319]. Cohen and, later, Wilson provide broad surveys of the literature on garbage collection [99,362]. Conservative collectors were introduced by Boehm and Weiser [48,50,131]. Copying collectors appeared in response to virtual memory systems [86,154]; they led, somewhat naturally, to the generational collectors in widespread use today [255,349]. Hanson introduced the notion of arena-based allocation [189].

EXERCISES

Section 6.2

1. Show the call tree and execution history for the following C program:

```
int Sub(int i, int j) {
    return i - j;
}
int Mul(int i, int j) {
    return i * j;
}
int Delta(int a, int b, int c) {
    return Sub(Mul(b,b), Mul(Mul(4,a),c));
}
void main() {
    int a, b, c, delta;
    scanf("%d %d %d", &a, &b, &c);
    delta = Delta(a, b, c);
    if (delta == 0)
        puts("Two equal roots");
    else if (delta > 0)
        puts("Two different roots");
    else
        puts("No root");
}
```

Section 6.3

2. Some programming languages allow the programmer to use functions in the initialization of local variables but not in the initialization of global variables.
   a. Is there an implementation rationale to explain this seeming quirk of the language definition?
   b. What mechanisms would be needed to allow initialization of a global variable with the result of a function call?
3. The compiler writer can optimize the allocation of ARs in several ways. For example, the compiler might:
   a. Allocate ARs for leaf procedures statically.
   b. Combine the ARs for procedures that are always called together. (When α is called, it always calls β.)

   For each scheme, consider the following questions:

   a. What kind of calls might benefit from the optimization?
   b. How often do you expect such calls to occur in real programs?
   c. What is the impact of the optimization on runtime space utilization?

4. Consider the following definition for class Dog and class Retriever. Draw the structures that the compiler would need to create to support objects of type Retriever, defined as follows:

```
class Dog {
    private int Height;
    private int Weight;
    public int Sheds;

    public int GetHeight();
    public int GetWeight();
    public boolean PlaysFetch();
}

class Retriever extends Dog {
    private boolean Calm;
    public boolean Disposition();
    public boolean PlaysFetch();
}
```

   The PlaysFetch method in Dog simply returns false, while the PlaysFetch method in Retriever returns true. The Disposition method in Retriever consults the boolean Calm.

5. Consider the definitions of Dog and Retriever from the previous question.

   a. If a program casts an object of class Retriever to be an object of class Dog, an *upcast*, what differences in behavior occur? Operationally, how does this upcast work?

   b. If a program casts an object of class Dog to be an object of class Retriever, a *downcast*, what differences in behavior would you expect to occur? What problems might arise from a downcast?

   c. What do you think is a reasonable way for the language to handle attempted downcasts?

6. As an alternative to a global method cache, an implementation could maintain an inline cache at each call site—a single-entry cache to record the class and method most recently invoked from that site.

   Develop pseudocode to use and maintain such an inline method cache. Explain the initialization of the inline method caches and any modifications to the general method lookup routine required to support inline method caches.

Section 6.4

7. Consider the following program written in PASCAL-like pseudocode:

```
procedure main;
    var a : array[1...3] of integer;
        i : int;
    procedure p2(e : integer);
        begin
            e := e + 3;
            a[i] := 5;
            i := 2;
            e := e + 4;
            end;
    begin
        a := [1, 10, 77];
        i := 1;
        p2(a[i]);
        for i := 1 to 3 do
            print(a[i]);
        end.
```

Simulate its execution under call-by-value, call-by-reference, and call-by-name parameter binding rules. Show the results of the print statements in each case.

8. The use of call-by-reference parameters can create situations where two distinct parameters refer to the same memory location. Consider the following PASCAL procedure, with parameters passed by reference:

```
procedure mystery(var x, y : integer);
    begin
        x := x + y;
        y := x - y;
        x := x - y;
    end;
```

If the arithmetic does not produce either underflow or overflow:

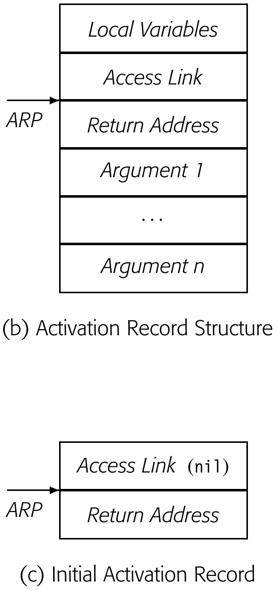
a. What result does `mystery` produce when it is called with two distinct variables, `a` and `b`?

b. What would be the expected result if `mystery` is invoked with a single variable `a` passed to both parameters? What is the actual result in this case?

9. Consider the PASCAL program shown in Fig. 6.9(a). Suppose that the implementation uses ARs as shown in Fig. 6.9(b). (Some fields have been omitted for simplicity.) The implementation stack allocates the ARs, with the stack growing toward the top of the page. The ARP

```
 1  program main(input, output);
 2      procedure P1(function g(b: integer): integer);
 3          var a: integer;
 4          begin
 5              a := 3;
 6              writeln(g(2))
 7              end;
 8      procedure P2;
 9          var a: integer;
10          function F1(b: integer): integer;
11              begin
12                  F1 := a + b
13                  end;
14          procedure P3;
15              var a: integer;
16              begin
17                  a := 7;
18                  P1(F1)
19                  end;
20          begin
21              a := 0;
22              P3
23              end;
24      begin
25          P2
26          end.
```

(a) Example PASCAL Program



(b) Activation Record Structure

(c) Initial Activation Record

■ **FIGURE 6.9** Program for Exercise 9.

is the only pointer to the AR, so access links are previous values of the ARP. Finally, Fig. 6.9(c) shows the initial AR for a computation.

a. For the example program in Fig. 6.9(a), draw the set of its ARs just prior to the return from function F1. Use the line numbers to specify return addresses. Draw directed arcs for access links. Show space for local variables and parameters. Label each AR with its procedure name.

b. Show the same state, assuming that the implementation uses a global display rather than access links.

Section 6.5

10. For each of the following actions, does that action belong in the precall, prolog, epilog, or postreturn sequence? Give a short justification for your answer.

```
procedure main            procedure f1(integer x, y)      procedure f2(integer q)
    integer a, b, c           integer v;                      integer k, r;
    b = a + c;                v = x * y;                      ...
    c = f1(a,b);              call print(v);                  k = q / r;
    call print(c);            call f2(v);                     end;
    end;                      return -x;
                              end;
```

■ **FIGURE 6.10** Code for Exercise 12.

a. store the caller-saves registers

b. restore the callee-saves registers

c. store the return value

d. store the return address

e. use the return address

11. In the design of a linkage convention, some actions can move between the precall sequence and the prolog, or between the epilog and the postreturn sequence.

    a. Why might it be good to move something from the precall sequence to the prolog? from the postreturn sequence to the epilog?

    b. What actions might you be tempted to move between these sequences?

12. Assume that the compiler is capable of analyzing the code to determine facts such as "*from this point on, variable* `v` *is not used again in this procedure*" or "*variable* `v` *has its next use in line 11 of this procedure*," and that the compiler keeps all local variables in registers for the three procedures shown in Fig. 6.10.

    a. Variable `x` in procedure `f1` is live across two procedure calls. For the fastest execution of the compiled code, should the compiler keep `x` in a caller-saves register or a callee-saves register? Justify your answer.

    b. Consider variables `a` and `c` in procedure `main`. Should the compiler keep them in caller-saves registers or callee-saves registers, again assuming that the compiler is trying to maximize the speed of the compiled code? Justify your answer.

This page intentionally left blank

Chapter 7

Code Shape

ABSTRACT

To translate an application program, the compiler must map each source-language statement into a sequence of one or more operations in the target machine's instruction set. The compiler must choose among many alternative ways to implement each construct. Those choices have a strong and direct impact on the quality of the code that the compiler eventually produces.

This chapter explores some of the implementation strategies that the compiler can employ for a variety of common programming-language constructs, with an emphasis on how to map the source-level constructs into the target machine's instruction set. Topics include expression evaluation, access methods for variables and aggregate data structures, control-flow constructs, and procedure calls.

KEYWORDS

Code Generation, Code Quality, Control Structures, Expression Evaluation

7.1 INTRODUCTION

When the compiler translates application code into executable form, it faces myriad choices about specific details, such as the organization of the computation and the location of data. Such decisions often affect the performance of the resulting code. The compiler's decisions are guided by information that it derives, progressively, over the course of translation. The compiler must encode that derived knowledge into the IR and its ancillary data structures. It encodes some of that knowledge directly in a transparent way. For example, if *x* is declared as a 64-bit integer, the compiler records that fact in the relevant symbol table.

Other facts are encoded implicitly. For example, if the compiler discovers that *y* is a local scalar variable and it decides that *y* is unambiguous, the compiler might encode that decision by keeping *y* in a virtual register throughout its lifetime. Encoding information about ambituity in this way makes the fact obvious to the rest of the compiler and eliminates the need to rederive this knowledge in later passes. We refer to this kind of implicit encoding of

Ambiguity can be difficult to determine. Some situations, such as an address-taken scalar variable, are obvious. Others, such as a C program that performs arithmetic on pointers, can require complex analysis.

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00013-9

knowledge into the IR program as *code shape*. The compiler writer makes an explicit decision to shape the IR in a specific way to encode information and, thus, expose it to later passes.

Conceptual Roadmap

The translation of source code into target-machine operations is one of the fundamental acts of compilation. The compiler must have a plan for each source-language construct. Many of the issues that arise when generating IR in the compiler's front end also arise when generating assembly code for a real processor in its back end. The target processor may present a more difficult problem due to finite resources and idiosyncratic features, but the principles are the same.

This chapter focuses on ways to implement a variety of source-language constructs in the compiler's IR. In many cases, specific implementation details affect the compiler's ability to analyze and to improve the code. The concept of code shape encapsulates all of the decisions, large and small, that the compiler writer makes about how to represent the computation in both IR and assembly code form. Careful attention to code shape can simplify the analysis and improvement of the code and can allow the compiler to produce better final code.

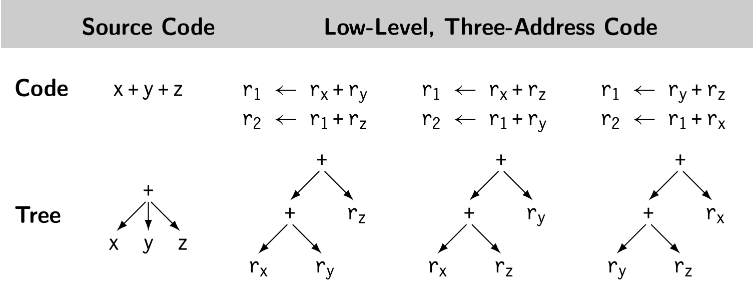
A Few Words About Time

At design time, the compiler writer must choose how the IR form of the program will express specific source-language constructs. Those decisions have a direct impact on the efficiency of the final compiled code. This chapter examines a number of source-language constructs and, when appropriate, explores some of the options available to the compiler writer.

Code-shape decisions, made by the compiler writer at design time, can determine the way that the compiler performs translation. To evaluate these choices, the compiler writer needs to consider the impact that the choices may have on later phases of the compiler and, eventually, on runtime performance. Thus, this chapter focuses on the impact that design-time decisions about the shape of the IR have on compile-time and runtime behavior.

Overview

In general, the compiler writer should focus on shaping the code so that later passes in the compiler can produce high-quality code. In practice, a compiler has multiple choices for how to implement many source-language constructs. These different options use different operations and different approaches. Some of these implementations are faster than others; some use



$x + y + z$

$r_1 \leftarrow r_x + r_y$
$r_2 \leftarrow r_1 + r_z$

$r_1 \leftarrow r_x + r_z$
$r_2 \leftarrow r_1 + r_y$

$r_1 \leftarrow r_y + r_z$
$r_2 \leftarrow r_1 + r_x$

■ **FIGURE 7.1** Alternate Code Shapes for x + y + z.

less memory; some use fewer registers; some might consume less energy during execution. We consider these differences to be matters of code shape.

Code shape has a strong impact on both the behavior of the compiled code and the ability of the optimizer and back end to improve the code. Consider, for example, the translation of a C `switch` statement that selects one of 256 cases based on a single-byte character. The compiler might implement the `switch` statement with a cascaded series of `if–then–else` statements. Depending on the layout of the tests, this could produce different results. If the first test is for zero, the second for one, and so on, then this approach devolves to linear search over the field of keys. If the tested characters are uniformly distributed, the corresponding searches will, on average, test half of the keys—an expensive way to implement a case statement.

On the other hand, the compiler might implement the `switch` statement as a binary search over the keys. In this scheme, the average case would test $log_2(|keys|)$, a more palatable number. To trade data space for speed, the compiler could construct a table of 256 labels and interpret the character by loading the corresponding table entry and jumping to it—with a constant overhead per character.

All of these are legal implementations of the `switch` statement. Choosing the best implementation for a specific `switch` statement depends on many factors. In particular, the number of cases and their relative execution frequencies play a critical role in the decision. Even when the compiler cannot determine these facts, it still must choose a strategy for each `switch` statement. The differences among the possible implementations, and the compiler's choice, are matters of code shape.

These three schemes are different enough that an optimizer is unlikely to convert one into another. The code-shape decision in translation will determine the final code shape for the switch statement.

As another example, consider the simple expression x + y + z, where x, y, and z are integers. Fig. 7.1 shows several ways to implement this expression. In source-code form, we may think of the operation as a ternary add, shown on the left. However, mapping this idealized operation into a sequence of

The figure assumes that each variable is in a register and that the source language does not specify the evaluation order for the expression.

binary additions exposes the impact of evaluation order. The three versions on the right show three possible evaluation orders, both as three-address code and as abstract syntax trees. If addition is commutative and associative, as integers are, all three orders are equivalent; the compiler must choose one.

Left associativity would produce the first binary tree. This tree seems "natural" in that left associativity corresponds to the left-to-right reading style of programming languages. Consider what happens if we replace y with the literal constant 2 and z with 3. Of course, x + 2 + 3 is equivalent to x + 5. The compiler should detect the computation of 2 + 3, evaluate it, and fold the result directly into the code. In both (x + y) + z and (x + z) + y, the code never evaluates 2 + 3. By contrast, (y + z) + x explicitly computes 2 + 3 and exposes the opportunity to replace the subtree with 5. For each prospective tree, however, there is an assignment of variables and constants to x, y, and z that does not expose the constant expression for optimization.

Again, the compiler cannot choose the best shape for this expression unless it understands the surrounding context. If, for example, the expression x + y was computed recently and the values of neither x nor y have changed, then using the leftmost shape would let the compiler replace the first operation, $r_1 \leftarrow r_x + r_y$, with a reference to the previously computed value. Often, the best evaluation order depends on context from the surrounding code.

This chapter explores the code-shape issues that arise in implementing many common source-language constructs. It focuses on the code that should be generated for specific constructs, while largely ignoring the algorithms required to pick specific assembly-language instructions. The issues of instruction selection, register allocation, and instruction scheduling are treated separately, in later chapters.

7.2 ARITHMETIC OPERATORS

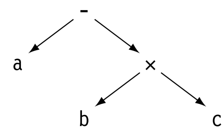
Modern processors provide broad support for evaluating expressions. A typical RISC machine has a full complement of three-address operations, including arithmetic operators, shifts, and Boolean operators. The three-address form lets the compiler name the result of any operation and preserve it for later reuse. It also eliminates the major complication of the two-address form: destructive operations.

To generate code for a trivial expression, such as a + b, the compiler first emits code to ensure that the values of a and b are in registers, say r_a and r_b. If a is stored in memory at offset @a in the current AR, the compiler might generate the code sequence shown in the margin to move a into register r_a.

loadI @a ⇒ r_1
loadAO r_{arp}, r_1 ⇒ r_a

```
expr(node) {
    int result, t1, t2;
    switch(type(node)) {

        case ×, ÷, +, -:
            t1 ← expr(LeftChild(node));
            t2 ← expr(RightChild(node));
            result ← NextRegister();
            emit(op(node), t1, t2, result);
            break;

        case NAME:
            entry  ← STLookup(node);
            result ← ValueIntoReg(entry);
            break;

        case NUMBER:
            num    ← NumberFromNode(node);
            result ← NumberIntoReg(num);
            break;
    }
    return result;
}
```

(a) Treewalk Code Generator



(b) Abstract Syntax Tree for a - b × c

```
loadI    @a          ⇒ r_1
loadAO   r_arp, r_1  ⇒ r_2
loadI    @b          ⇒ r_3
loadAO   r_arp, r_3  ⇒ r_4
loadI    @c          ⇒ r_5
loadAO   r_arp, r_5  ⇒ r_6
mult     r_4, r_6    ⇒ r_7
sub      r_2, r_7    ⇒ r_8
```

(c) Naive Code

■ **FIGURE 7.2** Simple Treewalk Code Generator for Expressions.

(Assume that r_{arp} holds the current ARP.) Once a and b are both in registers, the compiler can emit an add operation to compute their sum.

By contrast, if a's value already resides in a register, the code can simply reference its register. If both a and b already reside in registers, say r_a and r_b, then the compiler can simply emit the add operation.

As we saw in Section 5.3, the compiler can easily emit simple code for expressions in a syntax-driven translation framework. If the compiler initially builds a graphical IR, it can generate a linear form later in compilation using a simple treewalk, along the lines of the one shown in Fig. 7.2(a). It relies on several routines, denoted in the *algorithm* font, to emit code that instantiates the access method for a name or a number. (For more detail, see Section 7.3.)

Panel (b) shows a simple AST for a - b × c. Panel (c) shows the code that the treewalk code generator would produce for that AST. The example assumes that a, b, and c reside at known fixed offsets from the ARP—at offsets @a, @b, and @c.

GENERATING LOAD ADDRESS IMMEDIATE

A careful reader might notice that the treewalk code generator in Fig. 7.2 generates a two-operation sequence, `loadI` followed by `loadAO`, rather than a single `loadAI` operation:

$$\begin{array}{l} \texttt{loadI}\ \ \texttt{@a} \Rightarrow r_1 \\ \texttt{loadAO}\ r_{arp}, r_1 \Rightarrow r_2 \end{array} \quad \text{instead of} \quad \texttt{loadAI}\ r_{arp}, \texttt{@a} \Rightarrow r_2$$

Throughout the book, the examples assume that it is preferable to generate this two-operation sequence, rather than the single operation. Two distinct factors suggest this course.

1. The longer code sequence gives an explicit name to `@a`. If `@a` is reused in other contexts, that name can be reused.
2. The offset `@a` may not fit in the immediate field of a `loadAI`. That determination is best made in the instruction selector.

The compiler can convert the two-operation sequence into one operation during optimization, if legal and appropriate (e.g., either `@a` is not reused or it is cheaper to reload `@a`). The best course, however, may be to defer this choice to instruction selection—in effect, isolating the machine-dependent constant length into a phase of the compiler that is already highly machine dependent.

The simple treewalk code generator uses one template for all of +, -, ×, and ÷. It generates code to (1) ensure that the operands are in registers and (2) perform the operation, leaving the result in a new virtual register.

Most other three-address operations can fit into this framework. Some operations, such as exponentiation or a trigonometric function, require complex multioperation sequences for step (2). These may be expanded inline or implemented with a call to a library routine supplied by the compiler or the operating system.

7.2.1 Function Calls in an Expression

So far, we have assumed that all the operands in an expression are variables, constants, and temporary values produced by other subexpressions. Function calls also occur as operands in expressions. To evaluate a function call, the compiler simply generates the precall sequence needed to invoke the function, followed by the postreturn sequence, which moves the result into the appropriate location in the caller. The linkage convention limits the callee's impact on the caller.

The presence of a function call may limit the compiler's ability to reorder the evaluation of subexpressions. The function may have side effects that modify the values of variables used in the expression. The compiler must respect the implied evaluation order of the source expression, at least with respect to the call. Unless it knows about the call's possible side effects, the compiler cannot move references across the call. It must assume the worst case—that the function both modifies and uses every variable that it can access. The desire to improve on such worst-case assumptions motivated the development of interprocedural summary analysis (see Section 9.4).

7.2.2 **Mixed-Type Expressions**

Most programming languages allow the code to contain an operation that has operands of different types. These languages carefully define the meaning of such a "mixed-type expression." (We will focus on source-language base types rather than programmer-defined types.)

A typical language definition allows some mixed-type expressions, such as addition of an integer and a floating-point number, and disallows others, such as multiplication by a character string. In the allowed cases, the compiler may need to insert code that converts one or more of the values involved in the expression to another type.

The digression on page 248 summarizes the rules from the C++ standard.

The compiler writer has two choices with implicit conversions: make them explicit early in translation and expect optimization to reduce their cost, or keep them implicit until late in translation to reduce clutter in the IR.

For explicit conversions, the IR needs a way to represent them: appropriate nodes in a graphical IR or operators in a linear IR.

Inserting explicit conversions is simple; as the compiler translates an expression, it applies the conversion rules of the source language. Most of these rules can be expressed in an operator-specific table that gives result type as a function of the operand types. A simple conversion table for + appears in the margin. The compiler looks up the result type and emits code to convert operands, as necessary, to the result type. It then emits code to perform the operation in the result type.

+	**int**	**float**
int	`int`	`float`
float	`float`	`float`

Conversion Table for +

C and C++ provide base types for many widths of integer, each with a distinct range of numbers that it can represent. Both languages apply an implicit "widening" rule that converts operands to the narrowest type that can represent both operands. Thus, to add an 8-bit integer and a 64-bit integer, C converts the 8-bit integer to 64-bit form before performing the addition.

If the language does not allow compile-time determination of types, then checking and conversion must be done at runtime. The situation may require the addition of an explicit type field to the runtime representation of the value and a case analysis at runtime to check types, convert operands, and

Runtime type fields are often called *tags*.

COMMUTATIVITY, ASSOCIATIVITY, AND NUMBER SYSTEMS

The compiler can often take advantage of the algebraic properties of operations. Addition, multiplication, and Booleans (and, or, xor) are all commutative and associative. Thus, the compiler should recognize that for any given `a` and `b`, `a + b` and `b + a` compute the same value. Similarly, if the code contains `a + b + c` and `d + a + b`, the compiler should recognize that both contain the subexpression `a + b`. If the compiler evaluates expressions in left-to-right order, it will not see the common subexpression because it will compute the second expression as `(d + a) + b`.

When possible, the compiler should use commutativity and associativity to improve code quality. Reordering expressions can expose additional opportunities for improvement.

> *Due to finite precision, floating-point numbers represent a subset of the real numbers, one that does not preserve associativity. Thus, compilers should not reorder floating-point expressions unless the language specifically allows it.*

Consider the following example: computing `a - b - c`, for floating-point `a`, `b`, and `c`. We can assign values such that:

```
b, c < a        a - b = a        a - c = a
```

but `a - (b + c)` $\neq$ `a`. In this case, the result depends on the order of evaluation. Evaluating `(a - b) - c` produces a value indistinguishable from `a`, while evaluating `a - (b + c)` produces a value distinct from `a`.

This problem arises from the approximate nature of floating-point numbers; the mantissa is small relative to the range of the exponent. To subtract two numbers, the hardware must normalize them; if the difference in exponents is larger than the precision of the mantissa, the smaller number will be truncated to zero. The compiler cannot easily work around this issue, so it should, in general, avoid reordering floating-point computations.

evaluate the operator. Runtime checking incurs costs every time the code executes; compile-time checking incurs that cost once.

Assignment as an Expression

The compiler can treat assignment as an operator that has an unusual evaluation method. The expression on the assignment's right side evaluates to a concrete value, often called an *rvalue*. All of the techniques discussed for expression evaluation produce rvalues. The expression on the left side of an assignment evaluates to a location, or *lvalue*, which might be a memory address or a register. Section 7.3 discusses the computation of lvalues.

Rvalue
an expression evaluated to a value, as on the right side of an assignment

Lvalue
an expression evaluated to a place, as on the left side of an assignment

```
loadI   @a       ⇒ r1
loadAO  rarp, r1 ⇒ r2
loadI   @b       ⇒ r3
loadAO  rarp, r3 ⇒ r4
loadI   @c       ⇒ r5
loadAO  rarp, r5 ⇒ r6
mult    r4, r6   ⇒ r7
sub     r2, r7   ⇒ r8
```

(a) Code from Fig. 7.2(c)

```
loadI   @a       ⇒ r1
loadAO  rarp, r1 ⇒ r1
loadI   @b       ⇒ r2
loadAO  rarp, r2 ⇒ r2
loadI   @c       ⇒ r3
loadAO  rarp, r3 ⇒ r3
mult    r2, r3   ⇒ r2
sub     r1, r2   ⇒ r2
```

(b) Code After Register Allocation

```
loadI   @c       ⇒ r1
loadAO  rarp, r1 ⇒ r2
loadI   @b       ⇒ r3
loadAO  rarp, r3 ⇒ r4
mult    r2, r4   ⇒ r5
loadI   @a       ⇒ r6
loadAO  rarp, r6 ⇒ r7
sub     r7, r5   ⇒ r8
```

(c) Evaluate b × c First

```
loadI   @c       ⇒ r1
loadAO  rarp, r1 ⇒ r1
loadI   @b       ⇒ r2
loadAO  rarp, r2 ⇒ r2
mult    r1, r2   ⇒ r1
loadI   @a       ⇒ r2
loadAO  rarp, r2 ⇒ r2
sub     r2, r1   ⇒ r1
```

(d) Code After Register Allocation

■ **FIGURE 7.3** Rewriting a - b x c to Reduce Demand for Registers.

To generate code for an assignment, the compiler first emits code to evaluate the assignment's right-hand side to an rvalue. Next, the compiler evaluates the left-hand side to an lvalue, or location. If the rvalue and lvalue have different types, the language may require the compiler to insert code to convert the rvalue to the lvalue's type. Finally, the compiler emits code to move the rvalue into the specified location.

Evaluation of the lvalue may require code, as with an element of an array. In other cases, such as a scalar value kept in a register, it may not require code.

7.2.3 Reducing Demand for Registers

In general, the compiler can keep unambiguous scalar values in registers (see Section 4.7.2). A common code-shape strategy is to assign every value that can reside in a register to a virtual register and, then, to rely on the register allocator to decide which values actually reside in physical registers at each point in the code (see Chapter 13). The compiler can, however, make code-shape choices that reduce the register pressure in the final code.

Register pressure
The demand for registers is sometimes called *register pressure*.

Expression evaluation provides a clear example. Consider, again, the evaluation of a - b × c, with all of a, b, and c stored at known offsets from the ARP. Fig. 7.3 shows two different evaluation orders for the expression, both before and after register allocation. Panel (a) shows the code from Fig. 7.2(c), which assumed an unlimited set of virtual registers. Panel (b) shows that same code after register allocation; it requires three physical registers in addition to r_{arp}. This result makes sense; the code loads the values of a, b, and c into registers before it performs the multiply and the subtraction. Either a syntax-driven scheme or a left-to-right treewalk would produce this result.

Changing the evaluation order can lower the register pressure. Panel (c) shows a version of the code that evaluates the multiply operation first. It evaluates b × c before it loads a. After register allocation, this version requires one fewer physical register, as shown in panel (d).

The evaluation order in panel (a) corresponds to a left-to-right treewalk. The evaluation order in panel (c) corresponds to a right-to-left treewalk. Because multiply has higher precedence than subtraction, the left-to-right order loads a's value into a register, but must preserve it while it evaluates b × c. In this example, the right-to-left order defers the load of a until after the multiply.

Of course, the best order depends on context. At a binary operation, if the compiler needs more registers to evaluate one operand than it does for the other, it should evaluate the more demanding operand first. Doing so will reduce register pressure by one.

This observation is not new. Floyd observed the principle in 1961 [160]. Sethi applied it to reducing demand for registers a decade later [321].

To apply this rule, the compiler must examine the expression twice: the first time to compute demand and the second time to choose the order of subexpression evaluation and emit code. This observation applies in many optimization and code generation contexts. To obtain good results often requires that the compiler analyze the code before rewriting it.

SECTION REVIEW

Chapter 5 introduced the translation of expressions, along with a framework to generate either graphical or linear IRs. This section shows how the compiler can apply the same ideas in a traversal of the IR to generate a lower-level representation. It also explores some of the complications that arise in the IR for expressions, such as embedded function calls, type conversions, and assignment.

Building an IR for an expression and traversing it multiple times allows the compiler to analyze the code, derive knowledge about it, and adjust the code in ways that improve the efficiency of the final code. Because a significant portion of actual running time is spent in expression evaluation, efficient code for expressions is critical to the overall speed of the generated program.

REVIEW QUESTIONS

1. ILOC includes `rsubI`, an immediate reverse subtract. It can simplify expression translation. When might it be useful? Why not include the nonimmediate version, `rsub`, as well?
2. Describe two scenarios where multiple traversals of the IR can improve code quality.

7.3 ACCESS METHODS FOR VALUES

Section 7.2 implicitly assumes that a single access method works for all identifiers. In practice, different kinds of variables require different access methods. This section begins with an examination of access methods for scalar variables. It then describes the complications introduced by the need to find individual elements within aggregates, such as structures, objects, and arrays.

7.3.1 Access Methods for Scalar Variables

For scalar variables, the access method must first determine whether the value resides in a register or in memory. Register access is straightforward; the code simply uses the virtual register name. Access in memory requires different strategies depending on the variable's storage class.

Variables Stored in a Register

If the compiler can keep a variable's value in a register in the vicinity of a definition or a use, then it can directly use the register name in operations that reference the value. Access to register-based values is simple and immediate.

On most machines, direct access to an enregistered value provides same-cycle access for both uses and definitions. An operation such as `add` $r_1, r_2 \Rightarrow r_1$ can both read and write r_1 in the same cycle. One goal of optimization and the primary goal of register allocation is to keep values in registers in the regions where they are frequently used.

Variables Stored in Memory

A variable may have its primary location in memory for a number of reasons. The name may be ambiguous because the code has multiple names that might access the same location. The name may have global visibility, so that its ambiguity (or lack thereof) cannot be determined when compiling one part of the entire application. The name may have a static lifetime, so that its value must be preserved across different procedure invocations.

The IR can represent each variable in a lexically scoped language with its static coordinate: a pair $\langle l,o \rangle$ where l is the lexical level at which the variable is defined and o is its offset within the level l data area. Global and static variables are typically assigned level zero; for them, the compiler must also record their specific data area.

The general scheme to compute the address of a memory-based variable is to calculate *base* + *offset*, where *base* is the data area's virtual address and *offset* is the variable's distance from *base*. Different levels in the static coordinate require different strategies to compute *base*.

Local Variables

For local automatic variables, the compiler can allocate storage in the local data area, at one end of the procedure's AR. To reference the variable, the compiler emits code to add the variable's known offset to the current ARP, stored in the designated register r_{arp}.

Unambiguous, local, scalar, automatic variables can live in registers. If demand for registers is high, the register allocator may spill some of these values to memory for parts of their lifetimes. When it does, the values' spill locations are typically placed in the local data area so that the spill and restore code can use r_{arp} as a base address.

Local Variables of Surrounding Scopes

While the global scope surrounds the other lexical scopes, access to it uses a different mechanism.

For local automatic variables declared in lexically surrounding scopes, the compiler must emit code to use the addressability mechanism built and maintained by the linkage convention (see Section 6.4.3).

To find a level l variable using access links, the compiler emits code to find the level l access link, which serves as a base address. With a global display, it would load that base address (the ARP of the level l surrounding scope) directly. It then computes a virtual address as *base* + *offset* and uses it in the `load` or `store` operation.

Static and Global Variables

With global variables, the compiler might generate a unique name-mangled label for each such variable, eliminating the offset and the addition.

For a variable located in a static data area, the base address is marked with an assembly-level label—typically produced by mangling the name associated with the scope. The compiler obtains the textual label name for the reference and arranges to load it into a register as a relocatable assembly-level label.

Properties of the ISA will determine how the compiler obtains the runtime address for the textual label name. We will assume that the label's value fits into a load immediate operation. If that is not true, then the compiler may need to store the label in a static data area (a constant pool) associated with the procedure and load its value from that location. Of course, the code needs to locate the constant pool. One scheme is to locate the constant pool at a known distance from the start or end of the procedure and access it with a PC-relative address.

For an integer stored at offset 8 in a static data area, the code to compute the address might be:

```
loadI  <label> ⇒ r_i  // get the base address
addI   r_i, 8   ⇒ r_j  // add the offset
```

where < *label* > is the mangled label for the appropriate scope. This sequence can be followed by a load from r_j or a store to r_j.

Variables Passed as Parameters

Formal parameters present a conceptual challenge. Consider an integer function *f(x)* that takes a single integer parameter, *x*. Each call site might pass a different variable, or *actual parameter*, to *x*. Those actual parameters might have several different storage classes. The code to access *x* inside *f* must work independently of the actual parameter's storage class.

Many linkage conventions designate registers for the first several parameters. For call-by-value parameters, the compiler stores the actual value in the register; the compiler then uses the register name in operations that reference the formal parameter. For call-by reference parameters, the compiler stores the actual parameter's address in the register; it then emits `loads` and `stores` to access the formal parameter's value.

Note that call-by-reference parameters require one more indirection than do call-by-value parameters.

For parameters passed in memory, the compiler can emit code that uses the ARP as a base address. Each memory-based parameter has a unique offset from the ARP; the compiler can emit code to compute the virtual address of that slot and use that address. Again, for a call-by-value parameter, the AR slot holds the value, while for a call-by-reference parameter, it holds the value's virtual address.

Variables Stored in the Heap

Heap-based variables typically lack both a base address and an offset. The heap allocation process returns a virtual address, which must be preserved to retain access to the entity. That virtual address, in turn, must be stored in either a variable or a field in some other heap-allocated entity.

Garbage collection relies on the fact that a heap-allocated entity cannot be accessed if the running program does not have a copy of its address.

If the compiled code contains arbitrary pointer arithmetic or buffer overflows, that assumption may be false.

To access an entity in the heap, the compiler must retrieve the entity's address, using the appropriate access method. Given that address, it can emit operations to manipulate its memory-based value.

Consider a local automatic variable `root` that points to a list of heap-allocated structures, each of which has a `next` pointer. To generate code to traverse the list, the compiler generates code to load the value of `root`. It uses that value as the address of the start of an instance of the structure. To move down the list, it adds the offset of `next` within the structure to the address of the current list element and loads the address of the next element.

7.3.2 Access Methods for Aggregates

Aggregate objects have more complex access methods than scalar objects because the compiler must emit code to locate specific elements within the aggregate. To access an element of a structure, object, or array, the compiler needs an address for the start of the aggregate object, an offset within the aggregate, and a type.

Structure Elements

The compiler determines how to find the structure's start address from a symbol table lookup using the search path for the context where the reference occurs. This part of the process is analogous to locating a scalar variable. The compiler uses the storage class to determine how to find the start address. It might be an offset from the current ARP or the ARP of a surrounding scope. It might be an offset from some static or global label. Alternatively, the start address might be stored in some variable that, itself, must be located.

Once the compiler has emitted code to ensure that the start address is in a register, it performs a lookup in the symbol table for the structure's internal scope to obtain offset and type information. It emits code to add the start address to the offset—to create an effective virtual address for the element. The type information lets the compiler emit the appropriate load operation or operations.

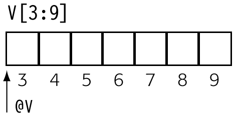
Object Members

To find a given member of a specific object, the compiler must first locate the object's OR. Because the running code uses the OR pointer (ORP) as the object's identity, finding the ORP, or start address of the OR, is straightforward. The source-language reference uses a name; that name contains the ORP. The compiler ensures that the ORP is in a register using the techniques described for a scalar variable.

To obtain the member's type and the offset in the OR, the compiler looks up the member name in the symbol table for the object's internal scope. It then emits code to add the OR address and offset, which creates the member's starting address. The type information lets the compiler emit the appropriate load operation or operations.

Vectors

Programming languages typically specify that vectors are one-dimensional arrays of contiguous memory, with a specified lower and upper bound on indices into the vector. A vector `V[3:9]` has (9 - 3) + 1 elements, as shown in the margin. To access the *i*th element of `V`, the compiler needs the starting address of `V`, which we will designate symbolically as `@V`, `V`'s lower and upper bounds, and the offset of element *i*.



Vector Layout

The compiler can compute the offset as $(i - low) \times w$, where *low* is the declared lower bound of `V` and *w* is the length of an element of `V`. Thus, if *low* is 3, *i* is 6, and *w* is 4, the offset from `@V` of the *i*th element is $(6 - 3) \times 4 = 12$. If $r_{@V}$ and r_i hold the values of `@V` and `i`, respectively, then the following code fragment loads the value of `V[i]` into $r_{V[i]}$.

```
subI   r_i,3      ⇒ r_1     // (i - lower bound)
multI  r_1,4      ⇒ r_2     // x element length (4)
add    r_@V,r_2   ⇒ r_3     // address of V[i]
load   r_3        ⇒ r_V[i]  // get the value of V[i]
```

This simple reference introduced three arithmetic operations. The compiler can do better. If the lower bound, *low*, is known, the compiler can fold the subtraction into `@V` to create $@V_0 = @V - low \times w$, a constant that we will call the *false zero* of `V`.

False zero
The false zero of a vector `V` is the address where `V[0]` would be.

In multiple dimensions, it is the location of a zero index in each dimension.

Many memory references take the form of a base address and an offset. Thus, most processors support a two-operand `load` operation similar to ILOC's `loadAO`, or load-address-offset, operation. Using `loadAO` folds the addition into the memory operation.

The third arithmetic operation is the multiply. On most processors, an integer multiply takes multiple cycles. If *w* is a power of two and the index, *i*, is positive, then the compiler can replace the multiply with a left-shift. Taken together, these would produce the ILOC sequence:

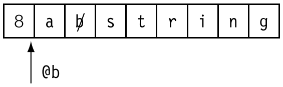
```
lshiftI  r_i,2         ⇒ r_2     // i × 4
loadAO   r_@V[0],r_2   ⇒ r_V[i]  // get value of V[i]
```

This code is shorter and, presumably, faster. A good assembly-language programmer might write this code.

The compiler writer may want to expose details of the address arithmetic to more general optimization, such as redundancy elimination or strength reduction (see Sections 8.4.1 and 10.7.2). Low-level improvements, such as converting the multiply to a shift and folding the add and the load can be performed late in compilation.

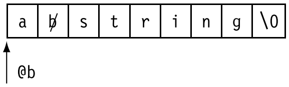
Strings

Strings are typically stored as a linear sequence of individual elements. Thus, the address calculation for an element in a string follows the same form as that for an element of a vector. String formats differ in the way that they represent the string's length. While the discussion focuses on character strings, other types of strings are similar.



String with Explicit Length Field

An explicit-length representation maintains an integer counter to contain the number of elements in the string, as shown in the margin. The drawing shows the length stored at a negative offset from the pointer to the string. To determine a string's length, the runtime code simply reads the value in the length field.



Null-Terminated String

An implicit-length representation uses a designated value to denote the end of the string. The best known implicit-length string format is C's null-terminated character string. As shown in the margin, it stores the null character, designated as "\0," as the final character in the string. To determine a string's length, the runtime code must walk the string and count characters up to the null character.

We will defer the implementation of string operations until Section 7.6, after the section on the implementation of loops and other control-flow operations.

Multidimensional Arrays

To access an element of a multidimensional array, the compiler also uses a base address plus offset scheme. In the multidimensional case, the offset calculation is more complex than for a vector or string. We show the calculation for a two-dimensional array stored in either row-major order or as a set of indirection vectors (see Section 5.6.3). The schemes generalize to higher dimensions. The code for column-major order follows a similar logic and is left as an exercise for the reader.

Row-Major Order. To describe address calculations for multidimensional arrays, we extend the notation from the vector access method. Consider an array `A` dimensioned as `A[1:2,1:4]`. Let low_1 refer to the first dimension's lower bound and low_2 refer to the second dimension's lower bound. For `A`,

$low_1 = 1$, $high_1 = 2$, $low_2 = 1$, and $high_2 = 4$. To simplify the exposition, let $len_k = high_k - low_k + 1$, the size of dimension k.

To access an element A[i,j], the compiler can emit code that (1) computes the offset for the start of row i; (2) computes the offset of element j within that row; and (3) computes the sum of the row offset, the column offset, and the base address. For A[i,j], that suggests the address computation:

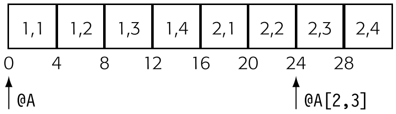
$$\mathtt{@A} + (\mathtt{i} - low_1) \times len_2 \times w + (\mathtt{j} - low_2) \times w$$

Here, @A is the base address of A; $(\mathtt{i} - low_1) \times len_2$ is the number of elements before the start of row i; and $(\mathtt{j} - low_2)$ is the offset (in elements) of the jth element of a row. The offsets are multiplied by w, the length in bytes of an element, to convert them to byte-addresses.

Substituting actual values for i, j, low_1, len_2, low_2, and w, we find that A[2,3] lies at offset

$$(2 - 1) \times (4 - 1 + 1) \times 4 + (3 - 1) \times 4 = 24$$

from A[1,1] (assuming that @A points at A[1,1], at offset 0). Looking at A's layout in memory, we find that the address of A[1,1] + 24 is, in fact, the address of A[2,3].



In the vector case, we were able to simplify the calculation when upper and lower bounds were known at compile time. Applying the same algebra to create a false zero in the two-dimensional case produces:

$$\mathtt{@A} + (\mathtt{i} \times len_2 \times w) - (low_1 \times len_2 \times w) + (\mathtt{j} \times w) - (low_2 \times w)$$

Reordering the terms produces the equation:

$$\mathtt{@A} + (\mathtt{i} \times len_2 \times w) + (\mathtt{j} \times w) - (low_1 \times len_2 \times w + low_2 \times w)$$

where the last term, $(low_1 \times len_2 \times w + low_2 \times w)$, is independent of i and j. This term can be factored directly into a false zero for A:

$$\mathtt{@A}_0 = \mathtt{@A} - (low_1 \times len_2 \times w + low_2 \times w) = \mathtt{@A} - 20$$

Now, the address calculation for A[i,j] is simply

$$\texttt{@A}_0 + \texttt{i} \times len_2 \times w + \texttt{j} \times w$$

Finally, we can refactor and move the w outside, saving a multiply

$$\texttt{@A}_0 + (\texttt{i} \times len_2 + \texttt{j}) \times w$$

For the address of `A[2,3]`, this evaluates to

$$\texttt{@A}_0 + (2 \times 4 + 3) \times 4 = \texttt{@A}_0 + 44$$

Since $\texttt{@A}_0$ is just $\texttt{@A} - 20$, this is equivalent to $\texttt{@A} - 20 + 44 = \texttt{@A} + 24$, the same location computed by the original address polynomial.

If we assume that `i` and `j` are in $\texttt{r}_\texttt{i}$ and $\texttt{r}_\texttt{j}$, and that len_2 is a constant, this refactored polynomial leads to the following code sequence:

```
loadI   @A_0          ⇒ r_{@A_0}  // adjusted base for A
multI   r_i, len_2    ⇒ r_1       // i × len_2
add     r_1, r_j      ⇒ r_2       // + j
multI   r_2, 4        ⇒ r_3       // x w
loadAO  r_{@A_0}, r_3 ⇒ r_a       // value of A[i,j]
```

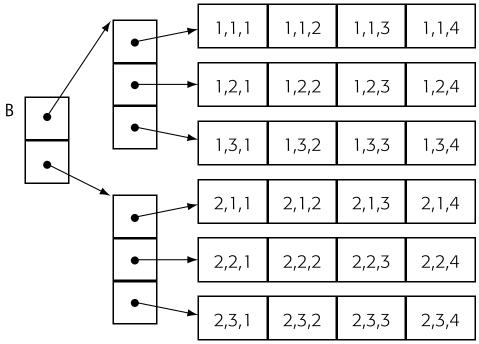
The computation is now two multiplications and two additions (one in the `loadAO`). The second `multI` can be rewritten as a shift.

The compiler could insert the false zero calculation after the procedure's prolog code and rely on dead code elimination to remove it if it is not needed—for example, if the only use of the array is as an actual parameter at a call site.

If the compiler does not have access to the array bounds, it must either compute the false zero at runtime or use the full polynomial that includes the lower bounds. The former option can be profitable if the elements of the array are accessed multiple times in a procedure; computing the false zero on entry to the procedure lets the code use the less expensive address computation. The more complex computation makes sense only if the array is accessed infrequently.

The ideas behind the address computation for arrays with two dimensions generalize to arrays of higher dimension. The address polynomial for an array stored in column-major order can be derived from the same ideas that underlie the address polynomial for row-major order. The optimizations that we applied to reduce the cost of address computations apply equally well to the address polynomials for these other kinds of arrays.

Indirection Vectors. The indirection vector representation is conceptually simple. Each level of the array uses a set of vectors, as shown in Fig. 7.4. The innermost level of vectors contains the actual data elements; successive outer levels contain pointers linked so that a series of vector accesses will lead to the desired array element.



■ **FIGURE 7.4** Indirection Vectors in Row-Major Order for B[1:2,1:3,1:4].

Consider the three-dimensional array B shown in Fig. 7.4. To access B[i,j,k], the compiler emits code that uses @B$_0$, i, and the length of a pointer to find the start of the indirection vector for the subarray B[i,*,*]. Next, it emits code to use that result, along with j and the length of a pointer, to find the start of the vector for B[i,j,*]. Finally, it uses that result, along with k and the length *w* of an element of B to compute the address of B[i,j,k].

If the values of i,j, and k exist in registers r_i,r_j, and r_k, respectively, and @B$_0$ is the zero-adjusted address of the first dimension, then the compiler might generate the following code to load B[i,j,k]:

```
loadI   @B0        ⇒ r@B0   // false zero of B
multI   ri, 4      ⇒ r1     // assume pointer is 4 bytes
loadAO  r@B0, r1   ⇒ r2     // get @B[i,*,*]

multI   rj, 4      ⇒ r3     // pointer is 4 bytes
loadAO  r2, r3     ⇒ r4     // get @B[i,j,*]

multI   rk, 4      ⇒ r5     // assume element length is 4
loadAO  r4, r5     ⇒ rb     // value of B[i,j,k]
```

This code assumes that the pointers in the indirection structure have been adjusted to account for nonzero lower bounds. If not, then the values in r_j and r_k must be decremented by the corresponding lower bounds. In the example, the multiplies can be replaced with shifts.

Using indirection vectors, the reference requires just two operations per dimension. On machines where memory access was fast relative to arithmetic—for example, most computers prior to 1985—this access method was fast. As the cost of memory accesses has increased relative to the cost of arithmetic, this scheme has lost its advantage in speed.

On cache-based machines, locality is critical to performance. When arrays grow to be much larger than the cache, storage order affects locality. Row-major and column-major storage schemes produce good locality for some array-based operations. The locality properties of an array implemented with indirection vectors are harder for the compiler to predict and, perhaps, to optimize.

We use the term *array of structure* to indicate that all of the array elements have the same structure type.

Arrays of Structure. Some programming languages allow the compiler to declare an array of structure—multiple copies of a structure or record with array-style indexes that locate specific instances (elements in the array of structure). The access method for an array of structure relies on the address polynomial for an array to find the start of the structure instance. Within the instance, the standard structure access method (base + appropriate offset) applies.

Accessing Array-Valued Parameters

Most languages allow an implementation to pass arrays as call-by-reference parameters to avoid the need to copy each element's value at a call. The caller and callee must agree on the array's layout and access method—the form of the address polynomial. The caller must supply the callee with the values that the access method needs.

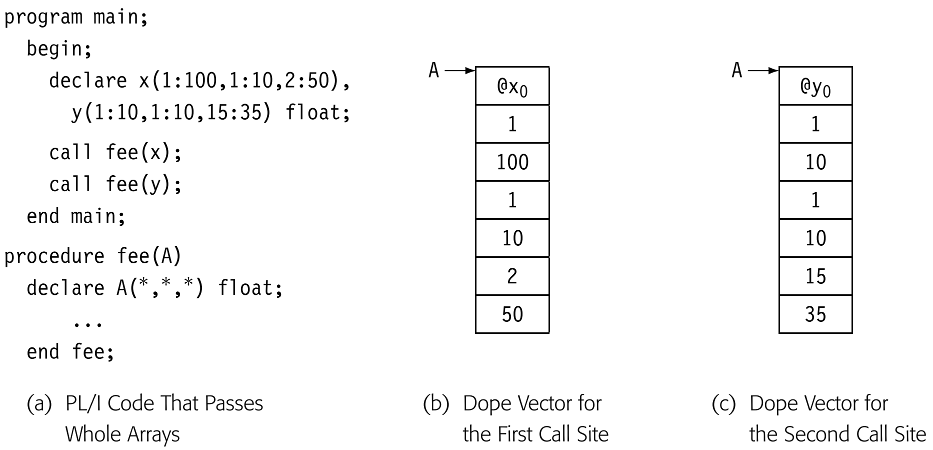
Dope vector
a descriptor for an actual parameter array

Dope vectors may also be used for arrays whose bounds are determined at runtime.

Some languages require the programmer to supply the needed values. For example, FORTRAN requires that the programmer declare the array using either constants or other formal parameters to specify its dimensions. Other languages have the compiler collect, organize, and pass the necessary information to the callee. The compiler builds a descriptor, called a *dope vector*, that contains both the array's base address and the necessary information for each dimension. The dope vector has a known size, so the compiler can allocate space for it in either the caller's AR or the callee's AR. The precall sequence passes the dope vector's address in the appropriate parameter slot.

Fig. 7.5 shows an example from PL/I. The procedure `main` invokes `fee` twice. Panels (b) and (c) show the dope vectors for the two call sites. If all the accesses used the false-zero version of the address polynomial, the dope vectors would contain the lengths of each dimension of `A` rather than the lower and upper bounds.

To access an element of the parameter array, the compiler emits the same address polynomial that it would use for a reference to a local array, except that it pulls values for the base address and dimensions from the dope vector. In this way, activations of the procedure from different call sites can use the same code to access distinct arrays passed from different call sites. The same technique can be used to support dynamically sized and allocated arrays. At



```
program main;
  begin;
    declare x(1:100,1:10,2:50),
      y(1:10,1:10,15:35) float;
    call fee(x);
    call fee(y);
  end main;
procedure fee(A)
  declare A(*,*,*) float;
     ...
  end fee;
```

(a) PL/I Code That Passes Whole Arrays

(b) Dope Vector for the First Call Site

(c) Dope Vector for the Second Call Site

■ **FIGURE 7.5** Dope Vectors.

allocation time, the compiler can construct a dope vector that subsequent references use.

Note that access through a dope vector costs more than access to a local array with known bounds. At best, the dope vector introduces additional memory references. At worst, it prevents the compiler from using optimizations that need facts from the array's declaration.

7.3.3 **Range Checks**

Most programming languages assume, either explicitly or implicitly, that a program only accesses locations that are within the defined bounds of data structures. For example, an array reference cannot access an element that is outside its lower and upper bounds. Neither can a string reference access elements that are outside its bounds.

A program that accesses an out-of-bounds element is, by definition, not well formed. Some languages, such as JAVA and ADA, require that out-of-bounds accesses be detected and reported. Other languages leave this issue to the compiler writer's discretion. Some compilers include code to perform these *range checks*; others do not. C's lack of checking is at least partially to blame for buffer-overflow attacks—arguably one of the most widespread computer security problems in history.

Range check
a test that verifies that an access to an aggregate data structure is within its declared or allocated bounds

The simplest implementation of a range check inserts a test before each reference to an aggregate entity. The test verifies that the computed index into the entity falls within its allocated range. The test throws an exception if the index is outside the valid range.

```
for $i \leftarrow 1$ to $n$
  for $j \leftarrow 1$ to $m$
    if ($lb_1 \leq i \leq ub_1$ and
        $lb_2 \leq j \leq ub_2$) then
          ... a[i,j] ...
    else throw exception
```

Naive Range Checking

```
if ($lb_1 \leq 1$ and $n \leq ub_1$ and
  $lb_2 \leq 1$ and $m \leq ub_2$)
  then
    for $i \leftarrow 1$ to $n$
      for $j \leftarrow 1$ to $m$
        ... a[i,j] ...
else throw exception
```

Range Checks After Optimization

The example in the margin inserts a test before the array reference in the inner loop. It assumes that a has bounds a[$\texttt{lb}_1$:$\texttt{ub}_1$,$\texttt{lb}_2$:$\texttt{ub}_2$]. A naive range check such as this one can create significant overhead. The additional control-flow affects both optimization and runtime.

The compiler can often improve naive checks by combining them or proving that the accesses are safe. In the marginal example, the checks in the inner loop can be combined and moved out of the two loops. The transformed code performs one check rather than one check per access.

If the loop nest contains multiple references to A, the impact might be even greater. With multiple identical references, the compiler can arrange to check the bounds once. If the loop nest contains multiple references with different index expressions, the compiler must cover all of them. It may be able to combine some of those tests algebraically, further reducing the cost.

If the compiler knows the value of some or all of the names used in the checks, it may be able to move some tests to compile-time. In the example, the compiler knows that the loop indices, *i* and *j*, start at one. If it knows the value of $\texttt{lb}_1$, it can perform the test $\texttt{lb}_1 \leq 1$ at compile time and eliminate it from runtime. The same strategy applies to each check; if the values are known, move the test to compile time.

For a heap-allocated object, the compiler could use the allocated size to approximate its total extents. This scheme may over-estimate the declared bounds, but would avoid references outside its allocated space.

To perform range checking, the compiler must know the data structure's size. If that size is static and known, the compiler can fold it directly into the check. With a dynamically sized data structure, the runtime system must record the size information that the range-check code needs.

SECTION REVIEW

The compiler needs a method to find and access each value that a program computes. The method may be trivial, as for a value kept in a register. It may require several instructions, as for a scalar value stored in the AR of a surrounding lexical scope. It may be complex, as for an element of an array of structures. For each kind of value and each type, the compiler needs a plan so that it can build the right IR construct for the source reference.

This section described common techniques to access most kinds of programming language values. A compiler will use a combination of these techniques, dictated by the implementation of other constructs and by the definition of the source language, itself. In practice, the translation should emit code that exposes enough detail to enable later passes in the optimizer and back end to generate compact and efficient code.

REVIEW QUESTIONS

1. Consider an array `A[0:99,0:89,0:109]`. Write down the code to compute the address of `A[i,j,k]` assuming (a) that `A` is stored in row-major order and (b) that `A` is stored using indirection vectors.
2. Explain, in terms of storage layout and the code required to access a specific element, the difference between an array of structure and a structure of arrays.

7.4 BOOLEAN AND RELATIONAL OPERATORS

Relational expressions typically produce Boolean values.

Programming languages almost always include features to specify control-flow, such as conditionals and loops. These features, in turn, create the need for Boolean and relational expressions. The language needs a grammar to specify these expressions. The compiler writer needs both a representation for Boolean values and a scheme to translate Boolean operations into the target machine's ISA.

The grammar uses $\vee$ for `or`, $\wedge$ for `and`, and $\neg$ for `not` to avoid confusion with ILOC operations.

The grammatical changes are straightforward, as shown in Fig. 7.6. Relational operators have higher precedence than Boolean operators. The productions for *RelExpr* allow Boolean combinations of relational expressions, but prevent cascading relational operators, as in $a < b < c$. The Boolean operators have their traditional precedence: $\neg$ then $\wedge$ then $\vee$. A relational expression produces a Boolean value.

The programming language may specify values for true and false, such as 1 and 0 or -1 and 0.

Representing Boolean values is also straightforward. The compiler writer can assign numerical values to true and false. Typical values include zero for false and nonzero for true. Well chosen values will allow the compiler to use the target machine's native operations to manipulate them. Most processors provide operations to perform negation, and, or, and exclusive or.

The complication in translating Boolean and relational expressions arises from the different kinds of support provided by processor ISAs. Section 7.4.1 introduces a basic scheme for translating these expressions. Section 7.4.2 examines how variations in the ISA support for Boolean and relational operations can affect translation.

7.4.1 Hardware Support for Relational Expressions

Processor ISAs provide several mechanisms to evaluate relational expressions. The operations available on the target ISA largely determine the kinds of code that the compiler can generate to evaluate a relational expression. Since relational expressions typically evaluate to a Boolean value, these

Expr	→	*Expr* ∨ *AndTerm*	*Term*	→	*Term Mults Value*
	|	*AndTerm*		|	*Value*
AndTerm	→	*AndTerm* ∧ *NotTerm*	*Value*	→	− *Factor*
	|	*NotTerm*		|	*Factor*
NotTerm	→	¬ *RelExpr*	*Factor*	→	$\underline{(}$ *Expr* $\underline{)}$
	|	*RelExpr*		|	num
RelExpr	→	*NExpr Rels NExpr*		|	name
	|	*NExpr*	*Rels*	→	$<$ | $\leq$ | $=$ | $\geq$ | $>$ | $\neq$
NExpr	→	*NExpr Adds Term*	*Adds*	→	$+$ | $-$
	|	*Term*	*Mults*	→	$\times$ | $\div$

■ **FIGURE 7.6** Adding Booleans and Relationals to the Expression Grammar.

effects carry over into both the representation and evaluation of Boolean operations, as well.

```
if (a<b) then
     statement_1
else statement_2
```

If-Then-Else Construct

Hardware support for relational expressions usually consists of a comparison operation and a set of operations that interpret the result of that comparison. While ISAs differ in how they provide this support, most of them use a similar scheme. Between the comparison operation and the operations that interpret it, the code must be able to distinguish the six basic relations: $<, \leq, =, \geq, >$, and $\neq$.

```
      cmp_LT   r_a, r_b ⇒ r_1
      cbr      r_1 → L_1, L_2
L_1:  statement_1
      jumpI    → L_3
L_2:  statement_2
      jumpI    → L_3
L_3:  nop
```

Implementation Scheme

Assume for the moment that, for each of these relations, the ISA includes a comparison operation that writes a Boolean value in a register as its result. For example, cmp_EQ $r_a, r_b \Rightarrow r_c$ sets r_c to true if the values in r_a and r_b are equal and to false otherwise. To interpret this value, a conditional branch operation, cbr $r_c \rightarrow L_T, L_F$, transfers control to the code at assembly label L_T if r_c contains true and to L_F otherwise.

The example in the margin shows how the compiler might implement a simple if–then–else construct using these operations. The code assumes that the values of a and b reside in r_a and r_b. The comparison operation, cmp_LT writes a Boolean value into r_1. The conditional branch, cbr, interprets the value in r_1 to choose between labels L_1 and L_2. After executing the appropriate *statement*, the code branches to the next statement, at L_3.

The compiler can implement Boolean operations, such as ∧, ∨, and ¬, in a manner similar to the arithmetic operations. Almost all processors provide direct hardware implementations of these operations, making the translation straightforward.

```
      cmp_LT   r_a, r_b ⇒ r_1
      cmp_GT   r_c, r_d ⇒ r_2
      and      r_1, r_2 ⇒ r_3
      cbr      r_3 → L_1, L_2
L_1:  statement_1
      jumpI    → L_3
L_2:  statement_2
      jumpI    → L_3
L_3:  nop
```

Evaluating (a < b ∧ c > d)

If the source construct contained a more complex condition, such as $(a < b \wedge c > d)$, the implementation could perform two comparison operations, combine their results with a hardware and operation, and branch on the result, as shown in the margin at the bottom of the previous page.

Short-Circuit Evaluation

In a Boolean expression, the value of one subexpression may determine the value of an entire expression. The code in the margin shows the ILOC that a compiler might emit for the assignment

$$x \leftarrow a < b \vee c < d \wedge e < f$$

assuming all six values are in appropriately named registers. The code evaluates each of the comparisons, then uses Boolean operations to combine those results. Assuming that cmp_LT, and, and or take one cycle each, the code requires five cycles to evaluate the expression.

```
cmp_LT  r_a, r_b ⇒ r_1
cmp_LT  r_c, r_d ⇒ r_2
cmp_LT  r_e, r_f ⇒ r_3
and     r_2, r_3 ⇒ r_4
or      r_1, r_4 ⇒ r_x
```

Complete Evaluation of
$x \leftarrow a < b \vee c < d \wedge e < f$

The compiler relies on two Boolean identities:

$$\forall x,\ \mathtt{false} \wedge x = \mathtt{false}$$
$$\forall x,\ \mathtt{true} \vee x = \mathtt{true}$$

To generate the short-circuit code, the compiler must analyze the expression in light of these two identities and find the set of minimal conditions that determine its value. In many cases, short-circuit evaluation can reduce the cost of evaluating Boolean expressions.

The compiler can refactor the code to use the minimal number of compares needed to determine the expression's value, as shown in the margin. It first computes $x \leftarrow a < b$; if x is true, it branches to the next statement in the source code, at L_3. On the other hand, if x is false, the code branches to L_1 to evaluate $x \leftarrow c < d$.

```
     cmp_LT  r_a, r_b ⇒ r_x
     cbr     r_x → L_3, L_1
L_1: cmp_LT  r_c, r_d ⇒ r_x
     cbr     r_x → L_2, L_3
L_2: cmp_LT  r_e, r_f ⇒ r_x
     jumpI   → L_3
L_3: nop
```

Code Refactored for
Short-Circuit Evaluation

If $c < d$ evaluates to false, then the x is false and the code branches to L_3 to execute the next source-code statement. If $c < d$ is true at this point, then the result of evaluating $e < f$ determines x's value; the code jumps to L_2 to perform that final evaluation.

The length of the path through the refactored code depends on the values of the three comparisons. If $a < b$, the expression evaluates to true and the code executes just one cmp_LT and one cbr.

If, instead, $a \geq b$, the result relies on the values of c, d, e, and f. If $c \geq d$, control jumps to L_3 and the evaluation takes two cmp_LTs and two cbrs. Finally, if $c < d$, the code uses three cmp_LTs, two cbrs, and a jump. (If the block for L_3 immediately follows the block for L_2, the jump should optimize away, leaving five operations, as in the original code.)

The actual costs will depend on the individual operation costs. On most machines, an untaken branch is a single cycle; a taken branch might be more. In many cases, the short-circuited code is faster than the code for the complete evaluation.

7.4.2 Variations in Hardware Support

Computer architects have introduced many variations in support for evaluation of Boolean and relational expressions. The compiler writer must understand the support that the target ISA provides and should capitalize on those features. This section examines three common variations on the support described in the previous section: conditional move operations, predicated execution, and condition codes.

Conditional Move Operations

To avoid some of the branches introduced by relational-expression evaluation, some ISAs include a *conditional move* operation. Conditional move takes two input values and assigns one of them to its result, based on either a Boolean value or the condition-code. If the conditional move operation takes an explicit Boolean argument, it will need to be a "four-address" operation, much like a floating-point multiply-add operation.

```
if (a<b) then
    x ← c + d
else x ← e + f
```

Conditional Move Example

```
add     rc, rd     ⇒ r1
add     re, rf     ⇒ r2
cmp_LT  ra, rb     ⇒ r3
c_i2i   r3, r1, r2 ⇒ rx
```

Corresponding ILOC Code Using Conditional Move

A conditional move provides a simple implementation of some if–then–else constructs, as shown in the margin. The compiler can emit code that evaluates both additions and only perform the assignment for the selected case—either the then part or the else part.

If the addition takes fewer cycles than the branch, this strategy saves runtime cycles. If the hardware can perform multiple adds in a single cycle, a good instruction scheduler may avoid paying an extra cycle for the add operation whose result is unused.

Predicated Execution

Predicated execution is another architectural feature intended to avoid the need for conditional branches. In this scheme, some or all operations can specify a predicate register that determines whether or not the operation takes effect. On some architectures, an operation with a false predicate does not execute; on others, it executes but does not apply its result. (That is, with a false predicate, an add does not assign its value and a branch does not transfer control.)

In ILOC, we designate a predicate as a register name enclosed in parentheses and followed by a question mark. The predicated add operation shown in the margin specifies that r_x receives the value $(r_a + r_b)$ if and only if r_1 contains the value true.

```
(r1)? add  ra, rb ⇒ rx
```

For the example used to demonstrate conditional move, a predicated execution scheme emits similar code, as shown in the margin. Again, architects can implement predication in two ways. The processor might evaluate all predicated operations and only assign the value if the predicate is true. Alternatively, the processor can test the predicate and only execute operations guarded by true predicates.

```
      cmp_LT  ra, rb ⇒ r1
      not     r1     ⇒ r2
(r1)? add     rc, rd ⇒ rx
(r2)? add     re, rf ⇒ rx
```

ILOC Emitted for Example Using Predicated Execution

Condition Codes

The discussion, so far, has assumed that the ISA's comparison operation returns a Boolean value. In practice, many processors return more fine-grained information, in the form of a *condition code*. In these ISAs, an operation such as compare r_a, r_b sets separate bits in a condition code to indicate whether $a < b$, $a = b$, or $a > b$. The condition code might be stored as a separate register; many architectures have stored it in the processor's status word. We can think of the condition code as an implicit argument that some operations define or use.

```
    comp    ra, rb ⇒ cc1
    cbr_LT  cc1 → L1, L2
L1: loadI   true  ⇒ r1
    jumpI   → L3
L2: loadI   false ⇒ r1
    jumpI   → L3
L3: nop
```

Deriving a Boolean Value from a Condition Code

Condition-code schemes require a conditional-branch operator that can test specific bits in the condition code. An ISA that uses condition codes probably has fewer comparison operations and more conditional branch operations than one with Boolean comparisons; the overall difference in complexity of the ISA is negligible.

```
if (a<b) then
    statement1
else statement2
```

If-Then-Else Construct

The complexity appears, however, in code that must produce a concrete Boolean value from a comparison, as shown in the margin. The compiler must emit code that uses a branch to interpret the condition code and embed assignments into the true and false paths. The comp operation writes its results into a condition-code register, cc_1; the conditional branch tests that value for the result "less than." The branch for $<$ writes the value true into the result, r_1, while the branch for $\geq$ writes the value false into r_1.

Naive translation of relational expressions into an ISA that uses condition codes can produce awkward code. The code from our earlier if–then–else example appears in the margin. It evaluates $a < b$ to a Boolean value and uses that value to select either the then part or the else part. The code is both awkward and long.

```
    comp    ra, rb ⇒ cc1
    cbr_LT  cc1 → L1, L2
L1: loadI   true  ⇒ r1
    jumpI   → L3
L2: loadI   false ⇒ r1
    jumpI   → L3
L3: loadI   true  ⇒ r2
    comp    r1, r2 ⇒ cc2
    cbr     cc2 → L4, L5
L4: statement1
    jumpI   → L6
L5: statement2
    jumpI   → L6
L6: nop
```

Naive Translation

The blocks at L_1 and L_4 execute if $a < b$, while the blocks at L_2 and L_5 execute if $a \geq b$. A human would simplify this code by moving *statement*$_1$ into the block at L_1 and *statement*$_2$ into the block at L_2. A compiler could achieve that result with a combination of constant propagation, code motion, and control-flow simplification (see Chapter 10). The combination of optimizations moves the two *statements*, empties blocks L_4 and L_5, removes those blocks, and redirects the jumps to L_6.

```
     comp     r_a, r_b ⇒ cc_1
     cbr_LT   cc_1 → L_1, L_2
L_1: statement_1
     jumpI     → L_6
L_2: statement_2
     jumpI     → L_6
L_6: nop
```

Implicit Representation for $a < b$

The resulting code, shown in the margin, represents the value of $a < b$ implicitly as a position in the code. Inside the block at L_1, that value is true; inside the block at L_2, it is false. The value is neither useful nor accessible in L_6 or afterward.

The compiler can emit code that represents the comparison's result implicitly. Compilers that target ISAs with condition codes often switch between explicit and implicit representations for these values. In general, the compiler can use an implicit representation unless the source code assigns the value to a variable. This approach eliminates the bulky and awkward code that naive translation might produce.

SECTION REVIEW

Processor ISAs implement and interpret comparisons in different ways. Translation schemes for Boolean and relational values reflect this diversity. This section showed a basic schema, along with several adaptations for different kinds of ISA support.

The compiler should make reasoned choices about how to map Boolean and relational expressions onto the ISA, choosing between implicit and explicit representations, and making use of features such as conditional moves and predication. Attention to these details can improve the efficiency of the generated code.

REVIEW QUESTIONS

1. Chapter 5 showed how to fit an `if-then-else` construct into a syntax-driven translation scheme (see page 224). Sketch a syntax-driven translation scheme to generate code for a relational expression, such as $a < b$, using the grammar from Fig. 7.6.
2. To include short-circuit evaluation into a translation scheme, the compiler must choose an order to evaluate an expression's subterms. How might the compiler choose an order that reduces the expected time for evaluation?

7.5 CONTROL-FLOW CONSTRUCTS

Programming languages provide control-flow constructs that allow the programmer to connect straight-line sequences of code, or basic blocks, in useful ways. Common control-flow constructs include an `if–then–else` construct, loop constructs, and case statements. For each control-flow construct, the compiler needs a translation scheme.

Recall that a basic block is just a maximal-length sequence of straight-line, unpredicated code.

Section 7.4 described the translation of `if–then–else` constructs. That scheme created three labels: one for the block that begins the `then` part, another for the block that begins the `else` part, and a third for the block that follows the `if–then–else` construct. The compiler can translate an `if–then–else` construct by building a representation for each block and connecting the blocks as dictated by the translation schemes for the `if–then–else` construct. Compilers handle other control-flow constructs in a similar fashion: building representations for the blocks and connecting them with labels, branches, and jumps.

To build a basic block, the compiler aggregates the code for consecutive, unlabeled, unpredicated operations. Any control-flow transfer ends the block. Any labeled statement ends the block, on the assumption that the label may be the target of a branch. Any predicated statement ends the block. In ILOC, every block begins with a labeled statement and ends with a branch or jump. In an IR that supports fall-through branches, the operation that follows a branch starts a block. It may lack an initial label.

To tie the blocks together, the compiler inserts code to implement the control-flow specified by the source program. The margin shows a source-level `if–then–else` construct, along with ILOC code that follows the discussion in Section 7.4.1. The example abstracts away the details of the blocks that form the `then` part and the `else` part.

```
if (a<b) then
    block1
else block2
```

If-Then-Else Construct

```
L0: cmp_LT  ra, rb ⇒ r1
    cbr     r1 → L1, L2
L1: block1
    jumpI   → L3
L2: block2
    jumpI   → L3
L3: nop
```

Implementation Scheme

The code to implement control-flow constructs resides at or near the end of the basic blocks. In general, blocks begin with labeled statements and end with branches or jumps. If the IR models delay slots, the branch or jump may be earlier in the block. In the ILOC examples that follow, blocks begin with a label. Since ILOC branches label both the taken and not-taken (or fall-through) cases, the first operation after a branch is only reachable if it is labeled. If the IR or the target ISA allow PC-relative branches or jumps, then a block may begin with an unlabeled statement that is the target of a PC-relative branch.

While languages use different syntax to express control flow, the set of underlying concepts is small. This section examines the implementation of conditionals, loops, and case statements.

BRANCH PREDICTION BY USERS

One urban compiler legend concerns branch prediction for `if–then–else` constructs. FORTRAN 66 had an arithmetic `if` statement that took one of three branches, based on whether the controlling expression evaluated to a negative number, to zero, or to a positive number. One early compiler allowed the user to supply a weight for each label that reflected the relative probability of taking that branch. The compiler then used the weights to order the branches to minimize the total expected delay.

After the compiler had been in the field for a year, the story goes, one of the maintainers discovered that the branch weights were being used in the reverse order, maximizing the expected delay. No one had complained.

This story is usually told as a fable about the value of programmers' opinions on the behavior of code they have written. (Of course, no one reported the improvement, if any, from using the branch weights in the correct order.)

7.5.1 Conditional Execution

The previous example showed a scheme to implement `if–then–else` constructs with a Boolean-valued compare and branches. The discussion in Section 7.4 focused on evaluating the controlling expression. It showed how the underlying instruction set influenced the strategies for handling the controlling expression.

Because the `then` and `else` parts can contain arbitrary amounts of code, the compiler writer may employ multiple translation strategies and select between them based on context. If the `then` and `else` parts are large, the three-block scheme from Section 7.4 works well. If, however, the `then` and `else` parts are small, the compiler should consider the use of either conditional moves or predicated execution to avoid the branches and jumps.

Predication decreases the density of useful operations in a block. At some point, the multicycle delay of a branch is less than the cost of the cycles spent on useless operations.

Consider an `if–then–else` construct where the `then` and `else` parts each contain 20 operations. With predication, the code would cause the processor to issue all 40 operations; with a branching strategy, it would issue 20 operations plus the necessary branches and jumps. For a small enough block, predication might be faster. As the block size grows, it will reach a point where the cost of the branches and jumps is less than the cost of issuing unused operations.

The same issue arises with a strategy based on conditional move operations. For small blocks, the use of conditional move operations may produce more efficient code than the three-block strategy.

The choice between strategies based on branches, predication, and conditional move operations should account for several factors.

1. *Expected Frequency of Execution* If one part of the `if–then–else` construct executes significantly more often, the compiler should focus on techniques that improve speed along that path. This bias might take the form of predicting a branch, of executing some operations speculatively, or of reordering the logic.
2. *Uneven Amounts of Code* If one part of the `if–then–else` construct contains many more operations than the other, this fact may weigh against the use of predication or conditional moves.
3. *Control Flow Inside the Construct* If either part of the `if–then–else` construct contains control flow, such as another `if–then–else`, a case statement, a loop, or a procedure call, then predication and conditional move implementations may be a poor choice.

To make the best decision, the compiler must consider all these factors, as well as the surrounding context. This observation suggests that, in practice, the compiler may need to (1) build an initial IR that represents the `if–then–else` in a more abstract way, (2) analyze the code in that form, and (3) choose between branches, predication, and conditional move after it has the context to make a good decision.

7.5.2 Loops and Iteration

Most programming languages include one or more loop constructs to perform iteration. The first FORTRAN compiler introduced the `do` loop to perform iteration. Today, loops are found in many forms. For the most part, they have a similar structure.

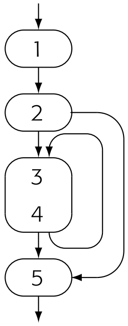
Consider the C `for` loop shown in Fig. 7.7(a) as an example. Panel (b) shows a CFG to illustrate how the compiler might lay out the code for this loop. The loop has three controlling expressions: e_1, which provides for initialization; e_2, which evaluates to a Boolean and governs execution of the loop; and e_3, which executes at the end of each iteration and, potentially, updates the values used in e_2. This basic schema applies to several kinds of loops.

If the loop body consists of a single block—that is, it contains no other control flow—then the schema produces a loop that has one initial branch plus one branch per iteration. Some ISAs provide mechanisms to hide branch latency, such as user-specified predictions or branch delay slots. Compilers can often use such features to hide the latency of a loop-closing branch.

If the ISA allows, the compiler should predict loop-closing branches as taken.

```
for (e_1; e_2; e_3) {
   loop body
}
```

(a) Code for Loop



Step	Purpose
1	*evaluate e_1*
2	*if ($\neg e_2$)* *then goto 5*
3	*loop body*
4	*evaluate e_3* *if (e_2)* *then goto 3*
5	*code after loop*

(b) Schema for Implementing the Loop

■ **FIGURE 7.7** General Schema for Layout of a `for` Loop.

For Loops

To map a `for` loop into code, the compiler follows the general schema from Fig. 7.7(b). Consider the following example:

```
for (i=1; i<=100; i++)
{
  loop body
}
next statement
```

```
      loadI   1      ⇒ r_i  // Step 1
      loadI   100    ⇒ r_1  // Step 2
      cmp_GT  r_i,r_1 ⇒ r_2
      cbr     r_2  → L_2,L_1
L_1:  loop body             // Step 3
      addI    r_i,1  ⇒ r_i  // Step 4
      cmp_LE  r_i,r_1 ⇒ r_3
      cbr     r_3  → L_1,L_2
L_2:  next statement        // Step 5
```

Translation of steps 1, 2, and 4 is straightforward. It yields a single basic block for steps 1 and 2, and another block for step 4. Translation of the loop body proceeds statement-by-statement. If the loop body consists of a single basic block, then the compiler can combine the blocks for steps 3 and 4, which may lead to improvements in the code. For example, the instruction scheduler might use operations from the end of step 3 to fill delay slots in the branch from step 4.

FORTRAN's DO Loop

In FORTRAN, the iterative loop is a `do` loop. It has a more restrictive form than C's `for` loop. The loop header specifies an index variable, an initial

SINGLE TEST VERSUS DOUBLE TEST LOOPS

The loop schema shown in Fig. 7.7 uses a compare-and-branch sequence both before the loop and at the end of the loop body. An obvious alternative is to adopt a schema that has a single test at the top of the loop and a jump at the bottom—avoiding the second compare operation.

The single-test loop creates a two-block loop body for even the simplest loops. It also lengthens the path through the loop by at least one operation.

The double-test loop should produce a single-block loop body for loops with no internal control flow. Local optimization and instruction scheduling then apply to the entire loop body. With constant lower and upper bounds, the initial check may be eliminated. In short, the double-test loop should optimize more easily than the single-test one.

value, and a final value. The programmer may specify an optional increment value. The following example loop iterates the index i from 1 to 100 by increments of 1.

```
    do 10 i = 1, 100, 1
       loop body
10     continue
    next statement
```

```
     loadI   1        ⇒ r_i  // Step 1
     loadI   100      ⇒ r_1  // Step 2
     cmp_GT  r_i, r_1 ⇒ r_2
     cbr     r_2 → L_2, L_1
L_1: loop body               // Step 3
     addI    r_i, 1   ⇒ r_i  // Step 4
     cmp_LE  r_i, r_1 ⇒ r_3
     cbr     r_3 → L_1, L_2
L_2: next statement          // Step 5
```

The comments map the ILOC code back to the schema in Fig. 7.7.

FORTRAN, like many languages, has some interesting quirks. One such peculiarity relates to do loops and their index variables. The number of iterations of a loop is fixed before execution enters the loop. If the program changes the index variable's value, or passes it to a subroutine that might change its value, that change does not affect the number of iterations that execute. To ensure the correct behavior, the compiler may need to generate a hidden index variable, called a *shadow index variable*, to control the iteration. For an index variable passed at a call, interprocedural summary analysis can prove that the value is unchanged, allowing the compiler to eliminate this extra overhead (see Section 9.2.4).

Programmers often pass a loop index at a call so that it can appear in the code's output. In FORTRAN, with call-by-reference parameters, this act can fool the compiler into believing that the index variable will be changed by the call.

While Loops

A `while` loop also maps into the standard loop schema. Unlike the C `for` loop or the FORTRAN `do` loop, a `while` loop has no initialization. Thus, the code is even more compact.

```
                            cmp_GE  r_x, r_y ⇒ r_1   // Step 2
                            cbr     r_1 → L_2, L_1
while (x < y) {
    loop body          L_1: loop body                // Step 3
}
                            cmp_LT  r_x, r_y ⇒ r_2   // Step 4
next statement              cbr     r_2 → L_1, L_2

                       L_2: next statement           // Step 5
```

Replicating the test in step 4 creates the opportunity for a simple loop to have a body that consists of a single basic block. The same benefits that accrue to a `for` loop from this structure also occur for a `while` loop.

Until Loops

An `until` loop iterates until the controlling expression has the value true. It tests the controlling expression at the end of each iteration of the loop's body. PASCAL's `repeat until` loop behaves in this way.

An `until` loop always enters the loop and performs at least one iteration. The loop has a particularly simple structure, since it avoids steps 1 and 2 in the schema:

```
{                      L_1: loop body                // Step 3
    loop body               cmp_LT  r_x, r_y ⇒ r_2   // Step 4
} until (x < y)             cbr     r_2 → L_2, L_1
next statement
                       L_2: next statement           // Step 5
```

C does not provide an `until` loop. Its `do` construct is similar to an `until` loop, except that the sense of the condition is reversed. It iterates until the condition evaluates to false, where an `until` loop iterates until the condition is true.

Expressing Iteration as Tail Recursion

In Lisp-like languages, programmers often implement iteration using a stylized form of recursion. If the last action executed by a function is a call, that call is known as a *tail call*. For example, to find the last element of a list in SCHEME, the programmer might write the following simple function:

Tail call
A procedure call that occurs as the last action in some procedure is termed a tail call. A self-recursive tail call is termed a *tail recursion.*

CONDITION CODES AND COUNTDOWN LOOPS

One argument for condition-code ISAs is that they can produce efficient code in some specific, but important, situations. In particular, these ISAs usually have arithmetic operations, such as `add` and `sub`, set the condition code to indicates the relationship between the result value and zero. In this case, subsequent control-flow operations can sometimes avoid the need to perform a comparison.

With this feature, the compiler can often translate an iterative loop with known bounds into a countdown loop that ends when an index variable reaches zero. (For example, a FORTRAN loop that needs a shadow index variable can be transformed to count down rather than up.) This transformation removes one operation from the loop. For loops that contain few operations, the improvement can be substantial.

```
(define (last alist)
    (cond
        ((empty? alist) empty)
        ((empty? (cdr alist)) (car alist))
        (else (last (cdr alist)))))
```

The recursive call in the last line is a tail call.

Compilers can generate particularly efficient code for tail calls that recur on the caller (see Section 10.4.1). Iteration can be expressed as a tail recursion, as in the following SCHEME code:

```
(define (count alist ct)
    (cond
        ((empty? alist) ct)
        (else (count (cdr alist) (+ ct 1)))))

(define (len alist)
    (count alist 0))
```

Invoking `len` on a list returns the list's length. `len` relies on `count`, which implements a simple counter using tail calls.

Break Statements

Many languages implement a `break` or `exit` statement that provides a structured way to exit a loop. A `break` transfers control to the first statement following the loop. For nested loops, a `break` typically exits the innermost loop. Some languages, such as ADA and JAVA, allow an optional label on a `break` statement that specifies the loop that it will exit. A labeled `break` allows the program to exit multiple loops at once. C also uses break in its switch statement, to transfer control to the statement that follows the switch statement.

The break statement has a simple implementation. Since loops and case statements end with a label for the next statement, the compiler can simply emit an immediate jump to that label. Some languages include a skip or continue statement that jumps to the next iteration of a loop. This construct can be implemented as an immediate jump to the code that reevaluates the controlling expression and tests its value. Alternatively, the compiler can simply insert a copy of the evaluation, test, and branch at the point where the skip occurs.

7.5.3 Case Statements

Many programming languages include some form of a case statement. FORTRAN has its computed goto. ALGOL-W introduced the case statement in its modern form. BCPL and C have a switch construct. PL/I generalized the switch statement; one version allowed logical expressions as case labels. As the introduction to this chapter hinted, implementing a case statement efficiently is complex.

Consider the implementation of C's switch statement. The basic strategy is simple: (1) evaluate the controlling expression; (2) branch to the selected case; and (3) execute the code for that case. Steps 1 and 3 are well understood; they follow from earlier discussions.

In C, the individual cases usually end with a break statement that exits the switch statement. If a case ends without a break, control falls through to the next case. To make this feature work, the compiler must either (1) preserve the original order of the cases, (2) add an explicit jump at the end of any case that lacks a break, or (3) clone the code for the fall-through case.

To make the case-statement implementation efficient, the compiler must choose a good method to find the desired case. The choice of case is not known until runtime and may, at runtime, change with each execution of the case statement. Thus, the compiler must emit code that will use the value of the controlling expression to locate the corresponding case. No single scheme works well for all instances. Many compilers have the ability to generate several different schemes and choose a specific scheme based on context.

This section examines three common strategies: a linear search, a computed address, and a binary search. Each strategy is appropriate under different circumstances.

```
switch ($e_1$) {
  case 0:  $block_0$;
           break;
  case 1:  $block_1$;
           break;
  case 3:  $block_3$;
           break;
  default: $block_d$;
           break;
}
```

(a) Switch Statement

```
$t_1$ ← $e_1$
if ($t_1$ = 0)
   then $block_0$
   else if ($t_1$ = 1)
      then $block_1$
      else if ($t_1$ = 3)
         then $block_3$
         else $block_d$
```

(b) Implementation with Linear Search

■ **FIGURE 7.8** Case Statement Implemented with Linear Search.

Linear Search

The simplest way to locate the appropriate case is to treat the case statement as the specification for a nested set of if–then–else statements. For example, the switch statement shown in Fig. 7.8(a) can be translated into the pseudocode shown in panel (b). This scheme preserves the meaning of the switch statement, but makes the cost of reaching individual cases dependent on the order in which they appear.

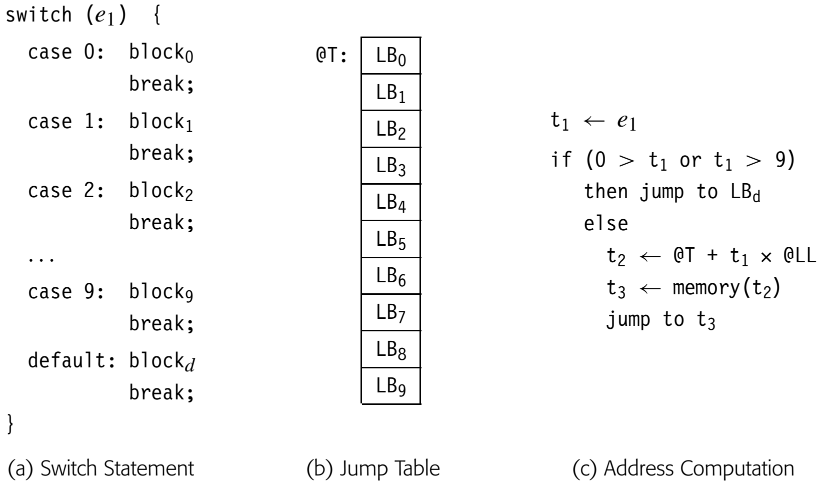
With a linear search strategy, the compiler should attempt to order the cases by estimated execution frequency. Still, when the number of cases is small—say three or four—this strategy can be efficient.

Direct Address Computation

Jump table
a vector of labels used to transfer control based on a computed index into the table

If the case labels form a compact set, the compiler can do better than linear search. Consider the switch statement shown in Fig. 7.9(a). It has case labels from 0 to 9, plus a default case. In this case, the compiler can build a vector, or *jump table*, that contains the block labels, and find the desired label by indexing into the table. The jump table is shown in panel (b), while the code to compute the correct case's label is shown in panel (c). The search code assumes that the jump table begins at @T and that a label occupies @LL bytes.

For a dense label set, this scheme generates compact and efficient code. The cost is small and constant—a brief calculation, a memory reference, and a jump. If a few holes exist in the label set, the compiler can fill those slots with the label for the default case. If no default case exists, the appropriate action depends on the language. In C, for example, the code should branch to the first statement after the switch, so the compiler can place that label in each hole in the table. If the language treats a missing case as an error, the



```
switch ($e_1$) {
  case 0:   block$_0$
            break;
  case 1:   block$_1$
            break;
  case 2:   block$_2$
            break;
  ...
  case 9:   block$_9$
            break;
  default:  block$_d$
            break;
}
```

(a) Switch Statement

@T:
LB$_0$
LB$_1$
LB$_2$
LB$_3$
LB$_4$
LB$_5$
LB$_6$
LB$_7$
LB$_8$
LB$_9$

(b) Jump Table

```
$t_1 \leftarrow e_1$
if ($0 > t_1$ or $t_1 > 9$)
   then jump to LB$_d$
   else
     $t_2 \leftarrow$ @T + $t_1 \times$ @LL
     $t_3 \leftarrow$ memory($t_2$)
     jump to $t_3$
```

(c) Address Computation

■ **FIGURE 7.9** Case Statement Implemented with Direct Address Computation.

compiler can fill holes in the jump table with the label of a block that throws a runtime error.

Binary Search

As the number of cases rises, linear search becomes inefficient. Similarly, as the label set becomes more sparse, jump table size becomes a problem for direct address computation. The classic solutions from efficient search apply here. If the compiler can impose an order on the case labels, it can use binary search to obtain a logarithmic search rather than a linear one.

The idea is simple. The compiler builds a compact ordered table of case labels, along with their corresponding block-address labels. It emits a binary search to find the case label or its absence. Finally, it either branches to the corresponding label or to the `default` case.

The exact form of the search loop might vary. For example, the code in panel (c) does not short circuit the case when it finds the label early. Empirical testing may be necessary to find the best choices.

Fig. 7.10(a) shows a case statement with a sparse set of labels (0, 15, 23, 37, 41, 50, 68, 72, 83, and 99) and a default case. The labels could, of course, cover a much larger range. Panel (b) shows the corresponding search table, and panel (c) shows a binary search that the compiler might emit to locate the desired case and branch to it.

To aid in subsequent analysis, the compiler should record the set of jump targets in the IR (such as ILOC's `tbl` pseudooperation described in Appendix A.5). Otherwise, the labels appear as data rather than code and passes that analyze control flow, such as the CFG construction, may have difficulty finding them (see Section 4.6).

```
switch (e1) {
  case 0:  block0
           break;
  case 15: block15
           break;
  case 23: block23
           break;
  ...
  case 99: block99
           break;
  default: blockd
           break;
}
```

(a) Switch Statement

Value	Label
0	LB_0
15	LB_{15}
23	LB_{23}
37	LB_{37}
41	LB_{41}
50	LB_{50}
68	LB_{68}
72	LB_{72}
83	LB_{83}
99	LB_{99}

(b) Search Table

```
t1 ← e1

down ← 0 // lower bound
up ← 10  // upper bound + 1

while (down + 1 < up)  {
  middle ← (up + down) ÷ 2
  if (Value[middle] ≤ t1)
    then down ← middle
    else up ← middle
}

if (Value [down] = t1
  then jump to Label[down]
  else jump to LBd
```

(c) Code for Binary Search

■ **FIGURE 7.10** Case Statement Implemented with Binary Search.

SECTION REVIEW

Programming languages include many features that implement control flow. The compiler needs a schema for each such construct in the source language. In some cases, such as a loop, one approach serves for multiple different constructs. In others, such as an if-then-else or a case statement, the compiler must choose between several implementation schemes.

To make the best choice in these more complex cases, the compiler writer may choose to build a more abstract initial IR and analyze that IR to determine the specific facts that the compiler needs to make an informed decision. Then, with those facts in hand, it can rewrite the construct to implement the chosen scheme.

REVIEW QUESTIONS

1. Write ILOC code for the FORTRAN loop shown in the margin. Recall that the loop body must execute 100 iterations, even though the loop modifies the value of i.
2. Many programming languages include an unconditional jump, such as the goto statement in C. How might the compiler manage the label for such a jump in a syntax-driven translation scheme?

```
   do 10 i = 1, 100
      loop body
      i = i + 2
10    continue
```

7.6 OPERATIONS ON STRINGS

FORTRAN 66 lacked support for character strings, a major omission.

Strings arise in multiple contexts. The most common strings contain characters, as found in most languages since COBOL. Some languages support strings with different element types. Basic string operations, however, have a common structure across many element types.

Support for string operations varies from that in C, where most manipulation takes the form of calls to library routines, to that in PL/I, where the language provides direct mechanisms to assign whole strings, substrings, and individual characters, and to concatenate strings to form new strings. To illustrate the translation issues that strings introduce, this section discusses string-length computation, string assignment, and string concatenation. Other string operations, such as substitution through a translation table, follow the same basic ideas.

String operations can be costly. Older CISC architectures, such as the IBM S/370 and the DEC VAX-11, provide extensive hardware support for character string manipulation. RISC machines rely more heavily on the compiler to code these complex operations using a set of simpler operations. The basic operation, copying bytes from one location to another, arises in many different contexts.

7.6.1 String Length

Programs that manipulate strings often need to know the length of a string. The implementation of a length operation depends on the underlying representation (see Section 5.6.3). For a string with an explicit length field, the compiler can simply emit code to access the length at a fixed offset from the start of the string (usually, a negative offset).

```
len ← 0
while (string[len] ≠ '\0')
    len ← len + 1
```

Finding the Length of a Null-Terminated String

For a null-terminated string, the running program must walk the string and count the elements, as shown in the margin. The pseudocode leaves the number of string elements in the variable `len`.

The length computation illustrates the tradeoff between the two representations. The explicit-length version uses more space and makes length an $\mathbf{O}(1)$ operation. The null-terminated version may use less space; it makes length an $\mathbf{O}(\textit{string length})$ operation.

7.6.2 String Assignment

String assignment breaks into two distinct cases: single-character assignment and multicharacter assignment. Single character assignment can be straightforward. In C, an assignment from the third character of `b` to the second character of a can be written as `a[1] = b[2];`. (Recall that C defaults to zero as the lower bound in a vector or string.) In an ISA with one-byte characters and byte-oriented memory operations (`cload` and `cstore`), this assignment might translate into the simple code shown in the margin. Assume that $r_{@a}$ and $r_{@b}$ hold the base addresses of a and b.

```
cloadAI   r@b, 2  ⇒ r2
cstoreAI  r2      ⇒ r@a, 1
```

Code for `a[1] = b[2];`

Without hardware-supported, byte-length, memory operations, the code becomes more complex. The compiler must emit code to load the word that contains `b[2]`. The code must use a mask to clear the rest of the word and then shift the character from `b[2]` into the position for `a[1]`. It must load the word that contains `a[1]` and clear that byte (another `and` of an appropriate mask), and move the character from `b[2]` into that position (with an `or` operation). Finally, it must write the word that holds `a[1]` back to memory. The complexity of this operation may explain why byte-length loads and stores are common.

The compiler can emit code that uses Boolean operations with appropriate masks to clear bytes or to move a byte from one word to another.

For multicharacter assignments, whether an entire string or a subrange, the compiler can wrap a loop around the code for a single-character assignment. With byte-length operations, the code is simple. The loop in the margin assigns all of b to a. It assumes that both strings have zero as a lower bound and that `size_of(a)` returns a's allocated length—that is, its capacity. To assign a subrange, the upper and lower bounds change on both strings.

```
la ← size_of(a)
lb ← length(b)
if (la < lb)
    then throw an error
i ← 0
while (i < lb) do
    a[i] ← b[i]
    i ← i + 1
```

Whole String Assignment

If the ISA lacks byte-oriented operations, multicharacter assignment can create complex situations. In the best case, the source and destination have the same memory alignment. In the worst case, the code must mask and shift each character to build the words in the destination string. These examples only arise in an ISA without byte-oriented memory operations and are left as an exercise for the reader.

To achieve efficient execution for long word-aligned strings, the compiler can generate code that uses whole-word loads and stores. Conceptually, the compiler wants to generate byte-oriented operations until it reaches a word or double-word alignment boundary, then use word or double-word operations as long as possible, and handle any remaining characters with byte-oriented operations. The effect is similar to unrolling the loop (see Section 8.5.2) and combining the operations, with a preloop to reach the alignment boundary and a postloop to handle any trailing characters.

As with single-character assignment, the code becomes more complex if the ISA lacks byte-oriented memory operations. In practice, the compiler's runtime library might include carefully optimized routines to implement the nontrivial cases (e.g., POSIX `memmove`).

7.6.3 String Concatenation

Concatenation is simply a shorthand for a sequence of one or more assignments. It comes in two basic forms: appending string `b` to string `a`, and creating a new string that contains `a` followed immediately by `b`.

If sufficient space is not available, the code should raise a runtime exception or signal an error.

The former case requires a length computation followed by an assignment. The emitted code determines the length of `a` and then, if space permits, it copies the characters from `b` into the space that immediately follows the contents of `a`. The latter case copies each character in `a` and each character in `b`. The compiler, in essence, treats the concatenation as a pair of assignments.

In either case, the compiler should ensure that enough space is allocated to hold the result. In practice, either the compiler or the runtime system must know the allocated length of each string. If the compiler knows those lengths, it can perform the check during code generation and avoid the runtime check. In cases where the compiler cannot know the lengths of `a` and `b`, it must generate code to compute the lengths at runtime and to perform the appropriate test and branch.

7.6.4 Optimization of String Operations

Because string operations often require O(*string length*) time, compiler writers should try to optimize them. A classic example of a string optimization problem is to find the length that would result from the concatenation of two strings, `a` and `b`. If + is the concatenation operator, we might write this expression as `length(a + b)`.

The expression `length(a + b)` has two obvious implementations: (1) construct `a + b` and compute its length (in C, `strlen(strcat(a,b))`), and (2) add the lengths of `a` and `b` (in C, `strlen(a) + strlen(b)`). The latter solution, of course, is desired, as it avoids creating a temporary string just to compute the length.

As discussed in Section 7.3.3, string operations may include a range check to avoid reading or writing characters outside of the string's allocated extent. Range checks can prevent logical errors from corrupting other data; they can also prevent malicious code from either reading or corrupting runtime state. The string assignment example shown earlier carefully placed the range check outside the loop.

Range checks, of course, require basic information about string length. To check a write, the test will need to know the allocated size of the string. To check a read, it may also need to know the current size of the string. (Consider a string allocated to have 1,024 bytes that currently holds the characters `"hello world!"`. An attempt to read the 128th character should trip a range check.) The same kinds of range-check optimizations that apply to arrays also apply to strings.

SECTION REVIEW

In principle, string operations are similar to vector operations. The details of string representation and the complications introduced by alignment and efficiency can complicate the code that the compiler must generate. Simple loops that copy one character at a time are easy to generate, to understand, and to prove correct. More complex loops that use word or double-word operations can be more efficient; the cost of that efficiency is added code to handle the corner cases. Many compilers handle the complex cases with a call to a string-copy routine in the runtime system.

REVIEW QUESTIONS

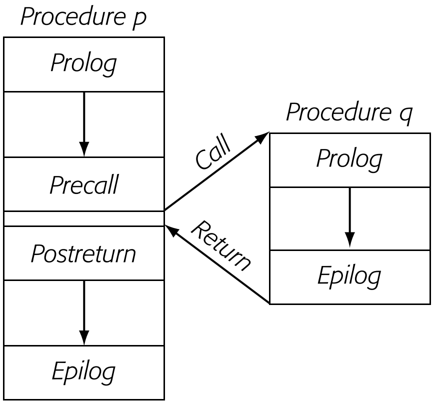
1. Write ILOC code for the string assignment `a` ← `b` using word-length loads and stores. (Use character-length loads and stores in a postloop to clean up the end cases.) Assume that `a` and `b` are word aligned and nonoverlapping.
2. If the two strings in question 1 are character aligned, what complications arise? Similarly, what complications arise if they may overlap?

7.7 PROCEDURE CALLS

The implementation of procedure calls is, for the most part, straightforward. As shown in Fig. 7.11, and discussed in Section 6.5, a call consists of four distinct sequences: the precall and postreturn sequences in the caller, and the prolog and epilog in the callee. A single procedure can contain multiple call sites, each with its own precall and postreturn sequences. In most languages, a procedure has one entry point, so it has one prolog sequence and one epilog sequence. Many of the details involved in these sequences are described in Section 6.5. This section focuses on issues that affect the compiler's ability to generate efficient, compact, and consistent code for procedure calls.

In languages that allow multiple entry points, such as PL/I, each entry point has its own prolog.

As a general rule, moving operations from the precall and postreturn sequences into the prolog and epilog sequences should reduce the overall size of the final code. If the call from *p* to *q* shown in Fig. 7.11 is the only call to *q* in the entire program, then moving an operation from the precall sequence



■ **FIGURE 7.11** A Standard Procedure Linkage.

in p to the prolog in q (or from the postreturn sequence in p to the epilog in q) has no impact on code size. If, however, other sites call q, then moving an operation from the caller to the callee (at all the call sites), it should reduce the overall code size by replacing multiple copies of an operation with a single one. As the number of call sites that invoke a given procedure rises, the savings grow. In general, this kind of linkage optimization can only be done during the design of the linkage convention, or in circumstances where the compiler knows and compiles each of the callers.

For example, a `static` function in C can only be invoked by procedures defined in the same file.

If the function's address is never taken, the compiler can customize its linkage.

We assume that most user-written procedures are called from several locations; if not, both the programmer and the compiler should consider moving the procedure inline at the point of its only call.

Procedure calls in Algol-like and object-oriented languages are similar. The primary difference between calls in ALLs and OOLs lies in the technique used to locate the code for the callee (see Section 6.3). In addition, a call in an object-oriented language typically includes the receiver's object record as an implicit parameter.

7.7.1 Evaluating Actual Parameters

When the compiler builds the precall sequence, it must emit code to evaluate the actual parameters to the call. The compiler treats each actual parameter as an expression. For a call-by-value parameter, the precall sequence evaluates the expression and stores its value in a location designated for that parameter—either in a register or in the callee's AR. For a call-by-reference parameter, the precall sequence evaluates the parameter to an address and stores the address in a location designated for that parameter. If a call-by-reference parameter has no storage location, then the compiler may need to allocate space in the caller's AR to hold the parameter's value so that it has an address to pass to the callee.

If the source language specifies an evaluation order for actual parameters, the compiler must implement that order. Otherwise, it should use a consistent order—either left to right or right to left. The evaluation order matters for parameters that might have side effects. For example, a program that used `push` and `pop` to manipulate a stack would produce different results for `subtract(pop(),pop())` under left-to-right and right-to-left evaluation.

Procedures typically have several implicit arguments, such as the procedure's ARP, the caller's ARP, the return address, and any information needed to establish addressability. Object-oriented languages pass the receiver as an implicit parameter. Some of these arguments are passed in registers while others usually reside in memory. Many architectures have an operation like

```
jsr label₁
```

that transfers control to $\texttt{label}_1$ and places the address of the operation that follows the `jsr` into a designated register.

Procedures passed as actual parameters may require special treatment. If *p* calls *q*, passing procedure *r* as an argument, *p* must pass to *q* more information than *r*'s starting address. In particular, if the compiled code uses access links to find nonlocal variables, the callee needs *r*'s lexical level so that a subsequent call to *r* can find the correct access link for *r*'s level. The compiler can construct an ⟨*address,level*⟩ pair and pass it (or its address) in place of the procedure-valued parameter. When the compiler constructs the precall sequence for a procedure-valued parameter, it must emit the extra code to fetch the lexical level and adjust the access link accordingly.

7.7.2 Saving and Restoring Registers

Under any calling convention, one or both of the caller and the callee must preserve register values. Often, linkage conventions use a combination of caller-saves and callee-saves registers. As both the cost of memory operations and the number of registers have risen, the cost of saving and restoring registers at call sites has increased, to the point where it merits careful attention.

In choosing a strategy to save and restore registers, the compiler writer must consider both efficiency and code size. Some processor features impact this choice. Features that spill a portion of the register set can reduce code size. Examples of such features include register windows on the SPARC machines, the multiword load and store operations on the Power architectures, and the high-level call operation on the DEC VAX-11. Each of these features offers the compiler a compact way to save and restore some portion of the register set.

While larger register sets can increase the number of registers that the code saves and restores, in general, using these additional registers improves the speed of the resulting code. With fewer registers, the compiler would be forced to generate loads and stores throughout the code; with more registers, many of these spills occur only at a call site. (The larger register set should reduce the total number of spills in the code.) The concentration of saves and restores at call sites presents the compiler with opportunities to handle them in better ways than it might if they were spread across an entire procedure.

- *Using Multiregister Memory Operations* When saving and restoring adjacent registers, the compiler can use a multiregister memory operation. Many ISAs support doubleword and quadword load and store operations. Using these operations can reduce code size; it may also improve execution speed.
- *Using a Library Routine* As the number of registers grows, so do the precall and postreturn sequences. The compiler may benefit from using custom save and restore routines with simplified call sequences. Across all calls, this strategy could produce a significant code-size savings.

  The save and restore routines can take an argument that specifies which registers must be preserved. It may be worthwhile to generate optimized versions for common cases, such as preserving all the caller-saves or callee-saves registers.
- *Combining Responsibilities* To reduce overhead further, the compiler might have the precall sequence pass the callee a value that specifies which caller-saves registers to preserve. The callee can then invoke the compiler's save routine to preserve both the caller and callee save registers in a single call. The epilog sequence can pass the same specification to the restore routine to reload the saved registers. This approach limits the overhead to a single call to save and another to restore. It separates responsibility (caller saves versus callee saves) from the cost to call the routine.

The compiler writer should pay close attention to the differences in code size and runtime speed among these distinct approaches. The code should use the fastest operations for saves and restores. This requires a close look at the costs of single-register and multiregister operations on the target architecture. Using library routines to perform saves and restores can save space; careful implementation of those library routines may mitigate the added cost of invoking them.

SECTION REVIEW

The code for a procedure call is split between the caller and the callee, and among the prolog, epilog, precall, and postreturn sequences. These sequences cooperate to implement the linkage convention, as discussed in Chapter 6. Language rules and parameter binding rules dictate both the order and the manner of evaluation for actual parameters. System-wide conventions determine responsibility for saving and restoring registers.

Compiler writers pay particular attention to the implementation of procedure calls because the optimization opportunities are difficult for general techniques to discover (see Chapters 8 and 10). The many-to-one nature of the caller-callee relationship complicates analysis and transformation, as does the distributed nature of the cooperating code sequences. Equally important, minor deviations from the defined conventions can cause incompatibilities in code from different compilers.

REVIEW QUESTIONS

1. When a procedure saves registers, either callee-saves registers in its prolog or caller-saves registers in a precall sequence, where do the values in those registers go? How much space should the compiler reserve to hold them?
2. In some situations, the compiler must create a storage location to hold the value of a call-by-reference parameter. What kinds of parameters may not have their own storage locations? What actions might be required in the precall and postreturn sequences to handle these actual parameters correctly?

7.8 SUMMARY AND PERSPECTIVE

One of the more subtle tasks that confronts the compiler writer is selecting a scheme to implement each source-language construct. Almost any source language construct can be implemented in multiple ways. The specific choices that the compiler writer makes at design time have a strong impact on the code that the compiler generates.

In a compiler that is not intended for production use—where the performance of the compiled code is less important—the compiler writer might select easy to implement translations that produce simple code. If the customers expect performance, as with an optimizing compiler, the compiler writer should focus on translations that expose as much information as possible to the later phases of the compiler—low-level optimization, instruction scheduling, and register allocation. These two different perspectives lead to different shapes for loops, to different disciplines for naming temporary variables, and, possibly, to different evaluation orders for expressions.

The classic example of this distinction is the `case` statement. In a debugging compiler, the implementation as a cascaded series of `if–then–else` constructs is fine. In an optimizing compiler, the inefficiency of the myriad tests and branches makes a more complex implementation scheme worthwhile. The effort to improve the `case` statement must be made when the IR is generated; few, if any, optimizers will convert a cascaded series of conditionals into a binary search or a direct jump table.

CHAPTER NOTES

The material contained in this chapter falls, roughly, into two categories: generating code for expressions and handling control-flow constructs. Expression evaluation is well explored in the literature. Discussions of how to handle control flow are more rare; much of the material on control flow in this chapter derives from folklore, experience, and careful reading of the output of compilers.

Floyd presented the first multipass algorithm for generating code from expression trees [160]. He points out that both redundancy elimination and algebraic reassociation have the potential to improve the results of his algorithm. Sethi and Ullman [321] proposed a two-pass algorithm that is optimal for a simple machine model; Proebsting and Fischer extended this work to account for small memory latencies [299]. Aho and Johnson [6] introduced dynamic programming to find least-cost implementations.

The predominance of array calculations in scientific programs led to work on array-address expressions and to optimizations (like strength reduction, Section 10.7.2) that improve them. The algebraic transformations used to simplify the address polynomial for multidimensional arrays follow Scarborough and Kolsky [317]. Dope vectors appeared in ALGOL-60 systems; Sattley describes their use to handle dynamically allocated arrays [316].

Harrison used string manipulation as a motivating example for discussing pervasive inline substitution and specialization [192]. The example with `length(a+b)` in Section 7.6.4 comes from that paper.

Mueller and Whalley describe the impact of different loop shapes on performance [280]. Bernstein provides a detailed discussion of the options that arise in generating code for case statements [43]. The best descriptions of linkage conventions and their implementations appear in processor-specific and operating-system-specific manuals.

EXERCISES

Section 7.2

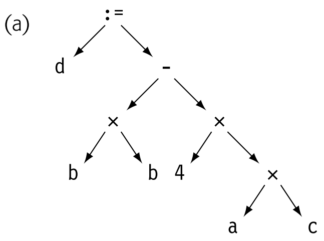
1. Extend the pseudocode for the treewalk code generator in Fig. 7.2 to insert type conversions for ×, ÷, +, and -. Assume that `type(node)` returns an enumerated value from the set

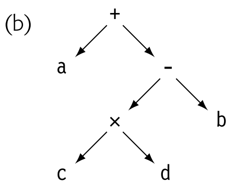
   ```
   { Short, Int, Float, Double }
   ```

   and that the conversions are specified by the table:

	Short	**Int**	**Float**	**Double**
Short	`Short`	`Int`	`Float`	`Double`
Int	`Int`	`Int`	`Float`	`Double`
Float	`Float`	`Float`	`Float`	`Double`
Double	`Double`	`Double`	`Double`	`Double`

2. For each of the trees below, show the ILOC code that the treewalk code-generation algorithm from Fig. 7.2 would emit. Assume an unlimited set of virtual registers.





3. For trees (a) and (b) above, show ILOC code that a translator might emit if it tried to minimize register use. For each interior node, indicate which of its children should be evaluated first.

Section 7.3

4. Implementation concerns often mean that textually identical references in the source program have different runtime costs. Compare and contrast the cost of reading the value of a local scalar variable x when:

   a. x is unambiguous

   b. x is ambiguous

   c. x is a call-by-value parameter of the current procedure

   d. x is a call-by-reference parameter of the current procedure

   e. x is static

5. For a character array `A[10:12,1:3]` stored in row-major order, show ILOC code to calculate the address of `A[i,j]`. Assume that `A` begins at `@A`. Use as few ILOC operations as you can.

6. Consider an integer array `A[1:7,2:9]` stored in column-major order (as in FORTRAN). Assume that `A` begins at `@A`.

   Derive the formula to compute the address of `A[i,j]` and write down ILOC code for the calculation.

Section 7.4

7. For the assignment `x` ← `a>b` ∧ `b>c`:

   a. Write ILOC to evaluate the statement using the ILOC comparison and conditional branch syntax (see page 763).

   b. Write ILOC to evaluate the statement using predicated operations, as shown in Section 7.4.

8. Consider a simple `if–then–else` construct:

```
if (x < y) then
    block1;
else block2;
```

   What properties of *block*$_1$ and *block*$_2$ should the compiler consider in choosing between an implementation with (a) a comparison and a conditional branch, (b) predicated execution, and (c) a conditional move? How does branch latency affect the decision?

Section 7.5

9. You recently implemented, by hand, a scanner. The scanner spends most of its time in a single `while` loop that contains a large case statement.

   a. How would the different case statement implementation techniques affect the efficiency of your scanner?

   b. How would you change your source code to improve the runtime performance under each of the case statement implementation strategies?

10. Convert the following C tail-recursive function to a loop:

```
List * last(List *l) {
  if (l == NULL)
     return NULL;
  else if (l->next == NULL)
     return l;
  else
     return last(l->next); }
```

   Show the C code for the loop.

```
do {
  *a++ = *b++;
} while (*b!='\0')
```

(a) Original Code

```
     loadI  @b      ⇒ r@b
     loadI  @a      ⇒ r@a
     loadI  NULL    ⇒ r1
L1:  cload  r@b     ⇒ r2
     cstore r2      ⇒ r@a
     addI   r@b, 1  ⇒ r@b
     addI   r@a, 1  ⇒ r@a
     cmp_NE r1, r2  ⇒ r4
     cbr    r4      → L1, L2
L2:  nop     // next statement
```

(b) Generated ILOC

■ **FIGURE 7.12** Example for Exercise 11.

Section 7.6

11. Consider the character copying loop shown in Fig. 7.12. Modify the ILOC code so that it branches to an error handler at L_{sov} on any attempt to overrun the allocated length of a. Assume that the allocated length of a is stored as an unsigned four-byte integer at an offset of –8 from the start of a.

12. String assignment on overlapping strings can become complex.

    a. Write the pseudocode for a function shuffle(a,s,l,o) that copies l characters starting at a[s] to a[s+o].
    Note that o can be positive or negative.

    b. How would you range check the assignment in shuffle? Assume that length(a) returns the number of characters stored in a and size_of(a) returns the allocated size of a.

Section 7.7

13. Consider an unambiguous, local, integer variable *x*. A call site in the procedure that declares *x* passes it as a call-by-reference actual parameter. The compiler can keep *x* in a register except at the call site. Inside the callee, *x* must have a memory address.

    a. Where should the compiler store *x*?

    b. How should the compiler handle *x* at the call site?

    c. How would your answers change if *x* was passed as a call-by-value parameter?

14. The linkage convention is a contract between the compiled code for a procedure and its callers. It creates a known interface that callers can use to safely invoke the procedure. Thus, the compiler should only violate

the linkage convention when such a violation cannot be detected from outside the compiled code.

a. Under what circumstances can the compiler safely choose to implement a variant (e.g., nonstandard) linkage? Give examples from real programming languages.

b. What changes might make the calling sequence more efficient and what facts would the compiler need to make those changes?

Chapter 8

Introduction to Optimization

ABSTRACT

To improve the quality of code that a compiler generates, its optimizer analyzes that code and rewrites it into a more efficient form. This chapter introduces the problems and techniques of code optimization and presents key concepts through a series of example optimizations. Chapter 9 expands on this material with a deeper exploration of program analysis. Chapter 10 provides a broader coverage of optimizing transformations.

This chapter examines how optimization can improve code. It discusses the safety and profitability of optimizations, along with the kinds of opportunities for improvement that a compiler writer should seek. It lays out the various scopes at which optimization is performed and presents two examples at each scope.

KEYWORDS

Optimization, Safety, Profitability, Scope of Optimization, Analysis, Transformation

8.1 INTRODUCTION

The compiler's front end translates the source-code program into some intermediate representation (IR). The back end translates the IR program into a form where it can execute directly on the target machine, either a hardware platform such as a commodity microprocessor or a virtual machine as in JAVA. Between these processes sits the compiler's middle section, its optimizer. The optimizer's task is to transform the IR program from the front end in a way that will improve the quality of the code that the back end produces. Our focus is on scalar optimization—that is, the optimization of single-threaded computations. Optimization of parallel programs is a complex and valuable subject that is beyond the scope of this book.

This chapter introduces the subject of code optimization and provides examples of several different techniques that attack distinct kinds of inefficiencies and operate on different regions in the code. Chapter 9 provides a deeper treatment of some of the techniques of program analysis that are used to support optimization. Chapter 10 describes additional code-improvement

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00014-0

transformations. Instruction scheduling, in Chapter 12, and register allocation, in Chapter 13, can be considered as problems in either optimization or code generation.

Conceptual Roadmap

The goal of code optimization is to discover, at compile time, information about the runtime behavior of the program and to use that information to improve the code that the compiler generates. Improvement can take many forms. The most common goal of optimization is to make the compiled code run faster. In some applications, however, the size of the compiled code outweighs its speed; consider, for example, an application that will be committed to read-only memory, where code size affects the cost of the system. Other optimization objectives include reducing the energy cost of execution, improving the code's response to real-time events, and reducing total memory traffic.

Optimizers use many different techniques to improve code. A proper discussion of optimization must consider the inefficiencies that can be improved and the techniques that can improve them. For each source of inefficiency, the compiler writer must choose from multiple techniques that try to address it; each technique has its own advantages and disadvantages. The compiler writer needs to understand how those techniques identify opportunities for improvement, how they determine that it is safe to rewrite the code, and why the rewrite represents an improvement. Each of these issues plays a role in successful code optimization.

Overview

Opportunities for optimization arise from many sources, but the primary one is the implementation of source-language abstractions. Because the translation from source code into IR is a local process—it occurs without extensive analysis of the surrounding context—it typically generates IR to handle the most general case of each construct. With more context, the optimizer can often determine that the code does not need that full generality; when that happens, it can rewrite the code in a more restricted and more efficient way.

A second significant source of opportunity for the optimizer lies with the target machine. The compiler must understand, in detail, the properties of the target machine that affect performance. Issues such as the number of functional units and their capabilities, the latency and bandwidth to various levels of the memory hierarchy, the various address modes supported in the instruction set, and the availability of unusual or complex operations all affect the shape of the code that the compiler should generate for a given application.

Optimization is a large and detailed subject whose study has filled complete courses and books. This chapter introduces the subject. Section 8.2 explores the concepts of *safety* and *profitability*; an optimization should only be applied when it is safe and it is expected to be profitable. Section 8.3 lays out the different granularities, or scopes, over which optimization occurs. The remainder of the chapter uses select examples to illustrate different sources of improvement and different scopes of optimization. This chapter has no Advanced Topics section; instead Chapters 9 and 10 serve that purpose. Chapter 9 delves into static analysis, describing the kinds of analytical problems that an optimizing compiler must solve and practical techniques that have been used to solve them. Chapter 10 examines scalar optimizations—those intended for a uniprocessor—in a more systematic way.

Safety
A transformation is *safe* when it does not change the results of running the program.

Profitability
A transformation is *profitable* to apply at some point when the result is an actual improvement.

A Few Words About Time

Chapter 8 provides an introduction to the subject of code optimization: both analysis and transformation. The optimizations described in this chapter occur entirely at *compile time*. Conceptually, each optimization reads the IR form of the program, analyzes it, transforms it, and writes it.

To decide whether or not to apply the optimization in a specific location in the code being compiled, the compiler will analyze the code. In essence, the compiler reasons about the runtime flow of values and control. It uses that information to decide whether it is safe to apply the transformation and whether the transformation is likely to improve the code. The compile-time analysis supports compile-time decision-making in order to improve runtime behavior.

8.2 BACKGROUND

Until the early 1980s, many compiler writers considered optimization as a feature that should be added to the compiler only after its other parts were working well. This fact led to a distinction between *debugging compilers* and *optimizing compilers*. A debugging compiler emphasized quick compilation at the expense of code quality. Debugging compilers did a simple translation, so the compiled code retained a strong resemblance to the source code. This correspondence simplified the task of mapping a runtime error to a specific line of source code; hence the term *debugging* compiler. By contrast, an optimizing compiler focused on improving the running time of the executable code at the expense of compile time. Spending more time in compilation can produce better code. With optimization, however, the mapping from source code to executable code is more complex and debugging might be, accordingly, harder.

OPTIMIZATION, ANALYSIS, AND TRANSFORMATION

The compiler construction community often uses the three terms *optimization*, *analysis*, and *transformation* as interchangeable. We make a distinction between these subtly different ideas.

Optimization refers to a broad scheme for improving program performance, such as code motion. Many algorithms exist for code motion. They differ in how they identify code that can move and how they choose its destination.

Analysis refers to compile-time techniques that derive knowledge about the runtime behavior of the compiled code. In code optimization, such analysis reveals opportunities where rewriting the code is safe and, potentially, profitable.

Transformation refers to compile-time techniques that rewrite the code to implement an optimization, usually based on the results of some analysis.

In some simple contexts, an optimization can be implemented in a single pass that combines analysis and transformation. In contexts that include complex control flow, the analysis typically must be completed before transformation can begin.

As RISC processors moved into the marketplace and as RISC implementation techniques were applied to CISC architectures, more of the burden for runtime performance shifted to compilers. To increase performance, processor architects turned to features that require more compiler support. These features include delay slots after branches, nonblocking memory operations, deeper pipelines, and more functional units. These features make processors more performance sensitive both to high-level issues of program layout and structure and to low-level details of scheduling and resource allocation. As the gap between processor speed and application performance has grown, optimization has become a routine part of every compiler.

The routine inclusion of an optimizer, in turn, changes the environment in which both the front end and the back end operate. Optimization further insulates the front end from performance concerns, which simplifies the task of translation in the front end. At the same time, optimization changes the code that the back end processes. Modern optimizers assume that the back end will handle resource allocation; thus, they typically target an idealized machine that has an unlimited supply of registers, memory, and functional units. This change, in turn, gives both more freedom and more responsibility to the compiler's back end.

If compilers are to shoulder their share of responsibility for runtime performance, they must include optimizers. As we shall see, the tools of optimization also play a large role in the compiler's back end. For these reasons, it is important to introduce optimization and explore some of the issues that it raises before discussing the techniques used in a compiler's back end.

8.2.1 **Examples**

To provide focus, we will explore two examples in depth. The first, a simple two-dimensional array-address calculation, shows the role that knowledge and context play in code optimization. The second, a loop nest from the routine `dmxpy` in the widely used LINPACK numerical library, provides insight into the transformation process itself and into the challenges that transformed code can present to the compiler.

Improving an Array-Address Calculation

Consider the IR that a compiler's front end might generate for an array reference `m(i,j)` in FORTRAN. Without specific knowledge about the values of `m`, `i`, and `j`, the compiler must generate the complete address polynomial for a two-dimensional array in column-major order:

$$\mathtt{@m} + (\mathtt{j} - low_2) \times (high_1 - low_1 + 1) \times w + (\mathtt{i} - low_1) \times w$$

The access method for an array in row-major order appears in Section 7.3.2.

where `@m` is the runtime address of the first element of `m`, low_i and $high_i$ are the lower and upper bounds of `m`'s ith dimension, and w is the size of an element of `m`. The compiler's ability to reduce the cost of this computation depends on its ability to analyze the code and its surrounding context.

The discussion assumes that the compiler has not transformed `@m` into the "false-zero" form $\mathtt{@m}_0$.

If `m` is a local array with $low_1 = low_2 = 1$ and the value of $high_1$ is known, then the compiler can simplify the calculation to

$$\mathtt{@m} + (\mathtt{j} - 1) \times hw + (\mathtt{i} - 1) \times w$$

where $hw = high_1 \times w$. If the reference occurs in a loop where `j` runs from 1 to k, the compiler might use *operator strength reduction* to replace the term $(\mathtt{j} - 1) \times hw$ with a sequence $\mathtt{j}'_1, \mathtt{j}'_2, \mathtt{j}'_3, \ldots, \mathtt{j}'_k$, where $\mathtt{j}'_1 = (1-1) \times hw = 0$ and $\mathtt{j}'_x = \mathtt{j}'_{x-1} + hw$. If `i` is also a loop induction variable and `i` runs from 1 to l, then strength reduction can replace $(\mathtt{i} - 1) \times w$ with the sequence $\mathtt{i}'_1$, $\mathtt{i}'_2$, $\mathtt{i}'_3$, $\ldots$, $\mathtt{i}'_l$, where $\mathtt{i}'_1 = 0$ and $\mathtt{i}'_x = \mathtt{i}'_{x-1} + w$. After these changes, the address calculation is just

$$\mathtt{@m} + \mathtt{j}' + \mathtt{i}'$$

Strength reduction
a transformation that rewrites a series of operations, such as

$$i \cdot c,\ (i+1) \cdot c,\ \ldots,\ (i+k) \cdot c$$

with an equivalent series

$$i'_0,\ i'_1,\ \ldots,\ i'_k,$$

where $i'_0 = i \cdot c$ and $i'_j = i'_{j-1} + c$

See Section 10.7.2.

Under the right circumstances, the addition in $\mathtt{@m} + \mathtt{j}'$ can be moved out of the inner loop (see Sections 10.3.1 and 10.7.2).

Code motion
moving a computation to a point where it executes less often

After these transformations, the j loop must increment j′ by *hw* and the i loop must increment i′ by *w*. In this form, the address computation consists of the increment to i′ and, perhaps, an add in the inner loop and the increment to j′ and an add in the outer loop. Knowing the context around the reference to m(i,j) allows the compiler to significantly reduce the cost of the array-element address expression.

If m is passed in as an actual parameter, then the compiler may not know its upper and lower bounds at compile time. In fact, the dimensions for m might change in different calls to the procedure. In such cases, the compiler may be unable to simplify the address calculation, another example of the role that context plays in performance.

Improving a Loop Nest in LINPACK

As a more dramatic example of context, consider the loop nest shown in Fig. 8.1. It is the central loop nest of the FORTRAN routine dmxpy from the LINPACK numerical library. The code wraps two loops around a single long assignment. The loop nest forms the core of a routine to compute $mx + y$, for a matrix m and vectors x and y. We will consider the code from two different perspectives: first, the transformations that the author hand-applied to improve performance, and second, the challenges that the compiler faces in translating this loop nest to run efficiently on a specific processor.

Before the author hand-transformed the code, the loop nest performed the following simpler version of the same computation:

```
      do 60 j = 1, n2
         do 50 i = 1, n1
            y(i) = y(i) + x(j) * m(i,j)
50       continue
60    continue
```

Loop unrolling
Unrolling replicates the loop body for distinct iterations and adjusts the index calculations to match.

To improve performance, the author *unrolled* the outer loop, the j loop, 16 times. That rewrite created 16 copies of the assignment statement with distinct values for j, ranging from j through j-15, and changed the outer loop increment from 1 to 16. Next, the author merged the 16 assignments into one statement, eliminating 15 occurrences of y(i) = y(i) + ···; that eliminates 15 additions and most of the loads and stores of y(i). Unrolling the loop eliminates 15 branches and some other operations. It may also improve cache locality.

To handle the cases where the array bounds are not integral multiples of 16, the full procedure has four versions of the loop nest that precede the one shown in Fig. 8.1. These “setup loops” process up to 15 columns of m,

```
      subroutine dmxpy (n1, y, n2, ldm, x, m)
      double precision y(*), x(*), m(ldm,*)
        ...
      jmin = j+16
      do 60 j = jmin, n2, 16
          do 50 i = 1, n1
             y(i) = ((((((((((((((( (y(i))
    $             + x(j-15)*m(i,j-15)) + x(j-14)*m(i,j-14))
    $             + x(j-13)*m(i,j-13)) + x(j-12)*m(i,j-12))
    $             + x(j-11)*m(i,j-11)) + x(j-10)*m(i,j-10))
    $             + x(j- 9)*m(i,j- 9)) + x(j- 8)*m(i,j- 8))
    $             + x(j- 7)*m(i,j- 7)) + x(j- 6)*m(i,j- 6))
    $             + x(j- 5)*m(i,j- 5)) + x(j- 4)*m(i,j- 4))
    $             + x(j- 3)*m(i,j- 3)) + x(j- 2)*m(i,j- 2))
    $             + x(j- 1)*m(i,j- 1)) + x(j) *m(i,j)
 50       continue
 60   continue
        ...
      end
```

■ **FIGURE 8.1** Excerpt from dmxpy in LINPACK.

leaving j set to a value for which n2 - j is an integral multiple of 16. The first loop handles a single column of m, corresponding to an odd n2. The other three loop nests handle two, four, and eight columns of m. The setup loops allow the final loop nest, shown in Fig. 8.1, to process 16 columns at a time.

Ideally, the compiler would automatically transform the original loop nest into this more efficient version, or into the most appropriate form for a given target machine. However, few compilers include all of the optimizations needed to accomplish that goal. In the case of dmxpy, the author performed the optimizations by hand to produce good performance across a wide range of target machines and compilers.

From the compiler's perspective, mapping the loop nest shown in Fig. 8.1 onto the target machine presents some hard challenges. The loop nest contains 33 distinct array-address expressions, 16 for m, 16 for x, and one for y. Unless the compiler can simplify those address calculations, the loop will be awash in integer arithmetic.

Consider the references to x. They do not change during execution of the inner loop, which varies i. The optimizer can move the address calculations and the loads for x out of the inner loop. If it can keep the x values in registers, it can eliminate a large part of the overhead from the inner loop. For a

Assume lower bounds of one for the arrays, which is standard in FORTRAN.

reference such as x(j-12), the address calculation is $@x + (j - 12 - 1) \times w$. To simplify further, the compiler can refactor all 16 references to x into the form $@x + jw - c_k$, where jw is $j \times w$ and c_k is $(k+1) \times w$ for each $0 \leq k \leq 15$. In this form, each load uses the same base address, $@x + jw$, with a distinct constant offset, c_k.

Mapping this calculation efficiently onto the target machine requires knowledge of the available address modes. If the target machine has a load operation with an address-immediate address mode (e.g., loadAI), then it can rewrite the accesses to x so that they use a single induction variable that starts at $@x + jmin \times w$; the j loop increments it by $16 \times w$.

Each iteration of the inner loop uses a different set of locations in m. Thus, the inner loop must compute new addresses and load new values for each of the 16 elements of m on each iteration. Careful refactoring of the address expressions, combined with strength reduction, can reduce the overhead of accessing m. The compiler can arrange to compute the address of m(1,j) in the j loop. It can then use distinct constant offsets in the load operations for each of the 15 other loads.

To achieve this code shape, the compiler must refactor the address expressions, perform strength reduction, recognize loop-invariant calculations and move them out of inner loops, and choose the appropriate address mode for the loads. Even with these improvements, the inner loop must perform 17 loads, 16 floating-point multiplies, and 16 floating-point adds, plus one store. The resulting block of code will challenge both the instruction scheduler and the register allocator.

Memory bound
A loop where loads and stores take more cycles than does computation is considered *memory bound*.

To determine if a loop is memory bound requires detailed knowledge about both the loop and the target machine.

If the compiler fails in some part of this transformation sequence, the resulting code might be substantially worse than the original. For example, if the compiler cannot refactor the address expressions around common base addresses for x and m, the code might need 33 distinct induction variables—one for each distinct location of x, m, and y. If the resulting demand for registers forced the register allocator to spill, that would insert more loads and stores into the memory-bound inner loop. In cases such as this one, the quality of code produced by the compiler depends on an orchestrated series of transformations that all must work; when one fails to achieve its purpose, the overall sequence may produce lower quality code than the user expects.

8.2.2 Considerations for Optimization

In dmxpy, the programmer applied the transformations by hand in the belief that they would make the program run faster. (They did.) The programmer also had to know that the changes would preserve the program's meaning.

DEFINING SAFETY

Correctness is the single most important criterion that a compiler must meet—the code that the compiler produces must have the same meaning as the input program. Each time the optimizer applies a transformation, that action must preserve the correctness of the translation.

Typically, *meaning* is defined as the program's observable behavior—that is, the state of its memory just before it halts, along with any output it generates. If the program terminates, the values of all visible variables immediately before it halts should be the same under any translation scheme. For an interactive program, behavior is more complex and difficult to capture.

Plotkin formalized this notion as *observational equivalence*.

> *For two expressions, M and N, we say that M and N are observationally equivalent if and only if, in any context C where both M and N are closed (that is, have no free variables), evaluating C[M] and C[N] either produces identical results or neither terminates [295].*

Thus, two expressions are observationally equivalent if their impacts on the visible, external environment are identical.

In practice, compilers use a simpler and looser notion of equivalence, namely, that if, in their actual program context, two different expressions *M* and *N* produce identical results, then the compiler can substitute *N* for *M*. This standard deals only with contexts that actually arise in the program. Tailoring code to context is the essence of optimization. It does not mention what happens when a computation goes awry, or diverges.

In practice, compilers take care not to introduce *divergence*; that is, they avoid creating code that either loops indefinitely or throws an exception such as division by zero. The opposite case, where the original code would diverge, but the optimized code does not, is rarely mentioned.

Safety, profitability, and risk each play a critical role in the compiler's decision to apply a given transformation at a specific location in the code.

The compiler needs a mechanism to prove that each application of the transformation is safe—that is, the transformation preserves the program's meaning. The compiler must have a reason to believe that applying the transformation is profitable—that is, the transformation will improve the program's performance. Finally, the compiler must understand the ways in which a rewrite might make the code worse—the risk that it is not profitable. The compiler writer needs to consider all three factors when designing and building the decision mechanisms that choose where to transform the code.

Safety

How did the LINPACK programmer know that the transformations were safe? That is, why did the programmer believe that the transformed code would produce the same results as the original code? Close examination of the loop nest shows that the only interaction between successive iterations occurs through the elements of `y`.

- A value computed as `y(i)` is not reused until the next iteration of the outer loop. The iterations of the inner loop are independent of each other, because each iteration defines precisely one value and no other inner-loop iteration references that value. Thus, the iterations can execute in any order. (For example, reversing the inner loop would produce the same results.)
- The interaction through `y` is limited in its effect. The *i*th element of `y` accumulates the sum of all the *i*th iterations of the inner loop. The unrolled loop reproduces this pattern.

A large part of the analysis done in optimization goes toward proving the safety of transformations.

Profitability

Why did the LINPACK programmer think that loop unrolling would improve performance? That is, why is the transformation profitable? Several different effects of unrolling might speed up the code.

- The number of loop iterations decreases by a factor of 16, which reduces the number of operations used to control the loop: adds, jumps, and branches. These savings can be significant.

  This effect might suggest unrolling by an even larger factor. Finite resource limits probably dictated the choice of 16. (See the discussion of *demand for registers* on the next page.)
- The array-address calculations contain duplicated work. Consider the use of `y(i)`. The original code computed `y(i)`'s address once per multiplication of `x` and `m`. The transformed code computes it once per 16 multiplications; it does $\frac{1}{16}$ as much work. The 16 references to `m`, and to a lesser extent `x`, also include common portions that the loop can compute once and reuse.
- The transformed loop performs more floating-point operations per memory operation. The original loop performed two floating-point operations per three memory operations; the unrolled loop performs 32 floating-point operations for 18 memory operations, assuming that all the `x` values reside in registers. Thus, the unrolled loop is less likely to be memory

The argument assumes that `x(j)` resides in a register.

bound. It has enough independent arithmetic to overlap the loads and hide some of their latencies.

Unrolling can help with other machine-dependent effects. It increases the amount of code in the inner loop, which may provide the instruction scheduler with more opportunities to hide latencies. If the end-of-loop branch has a long latency, the longer loop body may let the compiler fill more of that branch's delay slots. On some processors, unused delay slots must be filled with `nop`s, in which case loop unrolling can decrease the number of `nop`s fetched, reduce memory traffic, and, perhaps, reduce the energy used to execute the program.

If the inner loop grows larger than the code cache, performance may suffer.

Risk

If transformations intended to improve performance can make it harder for the compiler to generate good code, those issues should factor into the profitability decision. The hand transformations to `dmxpy` created new challenges for a compiler, including the following:

- *Demand for Registers* The original loop needs only a handful of registers to hold its active values. It needs floating-point registers for `y(i)`, `x(j)`, and `m(i,j)`. It needs general purpose registers for the address calculations for `x`, `y`, and `m`. The loop index variables need registers across loop iterations. By contrast, the transformed loop needs floating-point registers for the 16 elements of `x`, the 16 values of `m`, and one value of `y(i)`, along with the same general purpose registers.
- *Form of Address Calculation* The original loop deals with three addresses, one each for `y`, `x`, and `m`. Because the transformed loop refers to more distinct locations in each iteration, the compiler must shape the address calculations carefully to avoid repeated calculations and excessive demand for registers. In the worst case, the code could compute complete addresses for all 16 elements of `x`, all 16 elements of `m`, and one element of `y`.

  If the compiler carefully shapes the address calculations, it can use a single base address, or pointer, for `m` and another for `x`, each with 16 constant-valued offsets. It can rewrite the loop to use one of those pointers in the end-of-loop test, obviating the need for another register and eliminating another update (see Section 10.7.2). Planning and optimization make the difference.

Other problems of a machine-specific nature arise as well. For example, the 17 loads and one store, the 16 multiplies, the 16 adds, plus the address calculations and loop-overhead operations in each iteration must be scheduled with care. The compiler may need to use software pipelining to schedule some of the load operations so that they issue in a previous iteration to allow the initial arithmetic operations in the iteration to run in a timely fashion (see Section 12.5).

8.2.3 Opportunities for Optimization

As we have seen, the task of optimizing a simple loop can involve complex considerations. In general, optimizing compilers capitalize on opportunities that arise from several distinct sources.

1. *Reducing the Overhead of Abstraction* As we saw in `dmxpy`, data structures and types introduced by programming languages introduce runtime address calculations. Optimizers use analysis and transformation to reduce this overhead.
2. *Taking Advantage of Special Cases* Often, the compiler can use knowledge about the context in which an operation executes to specialize that operation. As an example, a C++ compiler can sometimes determine that a call to a virtual function always uses the same implementation. In that case, it can convert the virtual call to a less expensive static call.
3. *Matching the Code to System Resources* If a program's resource requirements differ from the processor's capacities, the compiler may transform the code to align its needs more closely with available resources. The transformations applied to `dmxpy` have this effect; they decrease the number of memory accesses per floating-point operation.

These are broad areas, described in sweeping generality. As we discuss specific analysis and transformation techniques, in Chapters 9 and 10, we will fill in these areas with more detailed examples.

SECTION REVIEW

Most compiler-based optimization works by specializing general purpose code to its actual context. For some transformations, the benefits accrue from local effects, as with the improvements in array-address calculations. Other transformations require broad knowledge of larger regions in the code and accrue their benefits from effects that occur over larger swaths of the code.

The implementation of any optimization must address three key issues. (1) How can the compiler efficiently find opportunities to apply the transformation? (2) Is the optimization safe in this context? (3) Will the optimization improve the code?

REVIEW QUESTIONS

1. In the code fragment from dmxpy, why did the programmer unroll the outer loop rather than the inner loop? How would the results differ had she unrolled the inner loop rather than the outer loop?
2. In the C fragment shown below, what facts would the compiler need to discover before it could improve the code beyond a simple byte-oriented, load/store implementation?

```
MemCopy(char *source, char *dest, int length) {
  int i;
  for (i=1; i≤length; i++) {
      *dest++ = *source++;
  }
}
```

8.3 SCOPE OF OPTIMIZATION

Optimizations operate at different granularities or scopes. In the previous section, we looked at optimization of a single array reference and of an entire loop nest. These different scopes presented different opportunities to the optimizer. Reformulating the array reference improved performance of the code for that one reference. Rewriting the loop nest improved performance across a larger region. In general, transformations and the analyses that support them operate on one of four distinct scopes: local, regional, global, or interprocedural.

Scope of optimization
The region of code that an optimization analyzes and transforms is its *scope of optimization*.

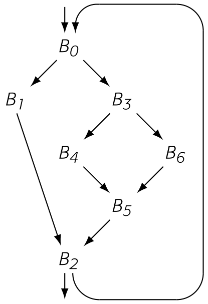
Local Methods

Local methods operate over a single basic block: a maximal-length sequence of branch-free code. In an ILOC program, a basic block begins with a labeled operation and ends with a branch or a jump. In ILOC, the operation after a branch or jump must be labeled or else it cannot be reached; other notations allow a “fall-through” branch so that the operation after a branch or jump need not be labeled. The behavior of straight-line code is easier to analyze and understand than is code that contains branches and cycles.

Inside a basic block, two important properties hold. First, statements are executed sequentially. Second, if any statement executes, the entire block executes, unless a runtime exception occurs. These two properties let the compiler prove, with relatively simple analyses, facts that may be stronger than those provable for scopes that contain conditional or cyclic control flow (if statements and loops). Thus, local methods sometimes make improvements that cannot be obtained for larger scopes. At the same time, local methods can only improve code sequences that occur entirely inside the same block.

Regional Methods

Regional methods operate over scopes larger than a single block but smaller than a full procedure. In the example control-flow graph (CFG) in the margin, the compiler might consider the entire loop, $\{B_0, B_1, B_2, B_3, B_4, B_5, B_6\}$, as a single region. In some cases, considering a subset of the code produces sharper analysis and better transformation results than would occur with information from the full procedure. For example, inside a loop nest, the compiler may be able to prove that a heavily used pointer is invariant (single-valued), even though it is modified elsewhere in the procedure. Such knowledge can enable optimizations such as keeping the value referenced through that pointer in a register throughout the loop nest.



Example CFG

The compiler can choose regions in many different ways. A region might be defined by some source-code control structure, such as a loop nest. The compiler might look at the subset of blocks in the region that form an *extended basic block* (EBB). The example CFG contains three EBBs: $\{B_0, B_1, B_3, B_4, B_6\}$, $\{B_5\}$, and $\{B_2\}$. While the two single-block EBBs provide no advantage over a purely local view, the large EBB may offer opportunities for optimization (see Section 8.5.1). Finally, the compiler might consider a subset of the CFG defined by some graph-theoretic property, such as a *dominator relation* or a strongly connected component in the CFG.

Extended basic block
a set of blocks $\{B_0, B_1, \ldots, B_k\}$ where either B_0 is the entry node or B_0 has multiple CFG predecessors, and every other B_i has just one predecessor, which is in the set

Dominator
In a CFG, x *dominates* y if and only if every path from the root to y includes x.

Regional methods have several strengths. Focusing on a region smaller than the entire procedure lets the compiler concentrate its efforts on heavily executed regions, such as the body of a loop. (A key observation in code optimization is that the body of a loop almost always executes more frequently than the surrounding code.) With regional methods, the compiler can apply different optimization strategies to distinct regions. Finally, the focus on a limited area in the code can let the compiler derive sharper information about program behavior which, in turn, may expose opportunities for improvement.

Intraprocedural Methods

These methods, also called *global methods*, use an entire procedure as context. The motivation for global methods is simple: decisions that are locally optimal may have bad consequences larger contexts. The procedure provides the compiler with a natural boundary for analysis and transformation.

INTRAPROCEDURAL VERSUS INTERPROCEDURAL

Few terms in compilation create as much confusion as the word *global*. Global optimizations operate on an entire procedure. The modern connotation, however, suggests an all-encompassing scope, similar to that of a global variable. In optimization, however, global means "pertaining to a single procedure."

As interest in analysis and optimization across procedure boundaries grew, the community needed terminology to differentiate between *global* analysis and analysis over larger scopes. *Interprocedural* was introduced to describe analysis that ranged from two procedures to a whole program. Accordingly, authors began to use *intraprocedural* for single-procedure techniques. Since these words are so close in spelling and pronunciation, they are easy to confuse and awkward to use.

One company, Perkin-Elmer Corporation, introduced the term *universal* to describe whole-program optimization. The term never gained popularity. We prefer the term *whole program* and use it whenever possible. It conveys the right distinction and reminds the reader that "global" is not "universal."

Procedures are abstractions that encapsulate and insulate runtime environments. They also serve as units of separate compilation in many systems.

Global methods typically operate by building a representation of the procedure, analyzing that representation, and transforming the underlying code. If the CFG has cycles, the compiler must analyze the entire procedure before it understands what facts hold on entrance to any specific block. Thus, most global optimizations have separate analysis and transformation phases. The analysis phase gathers facts and reasons about them. The transformation phase uses those facts to determine the safety and expected profitability of the transformations. By virtue of their global view, these methods can discover opportunities that neither local nor regional methods can.

Interprocedural Methods

These methods, sometimes called *whole-program methods*, consider scopes larger than a single procedure. We consider any transformation that involves more than one procedure to be an interprocedural transformation. Just as moving from a local scope to a global scope exposes new opportunities, so moving from single procedures to multiple procedures can expose new opportunities. It also raises new challenges. For example, name scope rules limit interactions between expressions in different procedures, and parameter binding complicates static analysis across procedure boundaries.

The term "whole program" clearly implies that the technique considers all of the code. We prefer the term "interprocedural" for techniques that analyze some, but not all, of the procedures and "whole program" for those that examine the entire program.

Interprocedural analysis and optimization occurs, at least conceptually, on the program's call graph. In some cases, these techniques analyze the entire program; in other cases, they may examine a subset of the program. Two classic examples of interprocedural optimizations are inline substitution, which replaces a call with a copy of the code for the callee, and interprocedural constant propagation, which propagates information about constants across the whole program.

SECTION REVIEW

Compilers perform both analysis and transformation over a variety of scopes, ranging from single basic blocks (local methods) to entire programs (whole-program methods). The set of available opportunities changes at different scopes; intuition suggests more opportunities in larger scopes. However, analyzing larger scopes often produces less precise knowledge about the code's behavior. Thus, no simple relationship exists between scope of optimization and code quality. It would be intellectually pleasing if larger optimization scopes led, in general, to better code. Unfortunately, that relationship does not necessarily hold true.

REVIEW QUESTIONS

1. Basic blocks have the property that if one operation executes, every operation in the block executes in a specified order (unless an exception occurs). State the weaker property that holds in an extended basic block.
2. The PL/I Optimizing Compiler was one of the first compilers to perform interprocedural analysis and optimization. However, it limited its scope to the set of procedures found in a single file. Justify this decision. What problems might arise if the compiler optimized a scope with multiple files?

8.4 LOCAL OPTIMIZATION

Local optimizations are among the simplest methods that the compiler can use. The simple execution model of a basic block, with serial execution of a single path, leads to reasonably precise analysis in support of optimization. Thus, these methods are surprisingly effective.

Redundant
An expression *e* is *redundant* at *p* if it has already been evaluated on every path that leads to *p*.

This section presents two local methods as examples. The first, *value numbering*, finds redundant expressions in a basic block and replaces the redundant evaluations with reuse of a previously computed value. The second, *tree-height balancing*, reorganizes expression trees to expose more opportunities to the scheduler.

8.4.1 Local Value Numbering

Consider the four-statement basic block shown in the margin. An expression, such as `b + c` or `a - d`, is redundant in a block if and only if it has been previously computed in the block and no intervening operation redefines one of its constituent arguments. In the example, the occurrence of `b + c` in the third operation is not redundant because the second operation redefines `b`. The occurrence of `a - d` in the fourth operation is redundant because the block does not redefine either `a` or `d` between the second and fourth operations.

```
a ← b + c
b ← a - d
c ← b + c
d ← a - d
```

Original Block

```
a ← b + c
b ← a - d
c ← b + c
d ← b
```

Rewritten Block

The compiler can rewrite this block so that it computes `a - d` once, as shown. The second evaluation of `a - d` is replaced with a copy from `b`. An alternative strategy would replace subsequent uses of `d` with uses of `b`. However, that approach requires analysis to determine whether or not `b` is redefined before each later use of `d`. In practice, it is simpler to have the optimizer insert a copy and let later passes determine whether or not the copy is needed (see the discussion of copy coalescing in Section 13.4.3).

Coalescing
a method that removes an unneeded copy, `a ← b`, by combining `a` and `b`

Lifetime
In the context of optimization, a value's lifetime is the region of code between its definitions and its uses.

In a block, a value's lifetime is the interval from its definition to its last use.

In general, replacing redundant evaluations with references to previously computed values is profitable—that is, the resulting code runs more quickly than the original. However, profitability is not guaranteed. Replacing `d ← a - d` with `d ← b` has the potential to extend the lifetime of `b` and to shorten the lifetimes of either `a` or `d` or both—depending, in each case, on where the value's last use lies. Depending on the precise details, each rewrite can increase, decrease, or leave unchanged the demand for registers. Replacing a redundant computation with a reference is likely to be profitable unless the rewrite causes the register allocator to spill additional values.

In practice, the optimizer cannot consistently predict the behavior of the register allocator, in part because the code will be further transformed before allocation. Therefore, most redundancy elimination algorithms assume that rewriting to avoid redundancy is profitable.

```
a ← b × c
d ← b
e ← d × c
```

Effect of Assignment

In the previous example, the redundant expression was textually identical to the earlier instance. Assignment can, of course, produce a redundant expression that differs textually from its predecessor. Consider the block shown in the margin. The assignment `d ← b` makes the expression `d × c` produce the same value as `b × c`. To recognize this case, the compiler must track the flow of values through names. Techniques that rely on textual identity do not detect such cases.

```
for i ← 1 to n, where the block has n operations in the form "$T_i$ ← $L_i$ $Op_i$ $R_i$" do
  get the value numbers $V_1$ and $V_2$ for $L_i$ and $R_i$
  if $T_i$ has no table entry then
      create a table entry for $T_i$
  construct a hash key, k, as ⟨$V_1$,$Op_i$,$V_2$⟩
  if k is already present in the table then
      replace operation i with "$T_i$ ← x", where x is the name stored under the key k
  else
      create a table entry for k
      generate a new value number for k and store it in k's table entry
      store the name $T_i$ in k's table entry
  store k's value number in the table entry for $T_i$
```

FIGURE 8.2 Value Numbering a Single Block.

The assignment to `y(i)` in Fig. 8.1 will create multiple redundant subexpressions from the various address polynomials.

Programmers will protest that they do not write code that contains redundant expressions similar to those in the example. In practice, redundancy elimination finds many opportunities. Translation from source code to IR elaborates many details, such as address calculations, and introduces redundant expressions.

Many techniques have been developed to find and eliminate redundancies. *Local value numbering* (LVN) is one of the oldest and most powerful of these transformations. It discovers redundancies within a basic block and rewrites the block to avoid them. It provides a simple and efficient framework for other local optimizations, such as constant folding and simplification using algebraic identities.

The Algorithm

The idea behind LVN is simple. The algorithm traverses a basic block and assigns a distinct number to each value that the block computes. It chooses the numbers so that two expressions, *M* and *N*, have the same value number if and only if *M* and *N* have provably equal values for all possible executions of the block.

Fig. 8.2 shows the basic LVN algorithm. LVN takes as input a block with *n* binary operations, each of the form $T_i \leftarrow L_i\ Op_i\ R_i$. LVN examines each operation, in order. It uses a hash table to map names, constants, and expressions into distinct value numbers. The hash table is initially empty.

To process the *i*th operation, LVN obtains value numbers for L_i and R_i by searching for them in the hash table. If it finds an entry, LVN uses the value

THE IMPORTANCE OF ORDER

The specific order in which expressions are written has a direct impact on the ability of optimizations to analyze and transform them. Consider the application of local value numbering to two distinct encodings of $v \leftarrow a \times b \times c$:

```
t0 ← a × b          t0 ← b × c
 v ← t0 × c          v ← a × t0
```

The encoding on the left assigns value numbers to a, b, $a \times b$, c, $(a \times b) \times c$ and v, while the encoding on the right assigns value numbers to b, c, $b \times c$, a, $a \times (b \times c)$ and v. Depending on the surrounding context, one of these may be preferable. For example, if $b \times c$ occurs later in the block but $a \times b$ does not, then the right-hand encoding produces redundancy while the left does not.

In general, using commutativity, associativity, and distributivity to reorder expressions can change the results of optimization. Similar effects can be seen with constant folding; if we replace a with 3 and c with 5, neither ordering produces the constant operation 3×5, which can be folded.

Because the number of ways to reorder expressions is prohibitively large, compilers use heuristic techniques to find good orderings for expressions. For example, the IBM FORTRAN H compiler generated array-address computations in an order that tended to improve other optimizations. Other compilers have sorted the operands of commutative and associative operations into an order that corresponds to the loop nesting level at which they are defined. Because so many solutions are possible, heuristic solutions for this problem often require experimentation and tuning to discover what is appropriate for a specific language, compiler, and coding style.

number of that entry. If not, it creates a new entry and assigns it a new value number.

Given value numbers for L_i and R_i, denoted V_1 and V_2, LVN constructs a hash key from $\langle V_1, Op_i, V_2 \rangle$ and looks up that key in the table. If an entry exists, the expression is redundant and can be replaced by a reference to the previously computed value. If not, then operation i is the first computation of L_i Op_i R_i in the block, so LVN creates an entry for its hash key and assigns that entry a new value number. It also assigns the hash key's value number, whether new or preexisting, to the table entry for T_i. Because LVN uses value numbers to construct the expression's hash key, rather than names, it can effectively track the flow of values through copy and assignment operations, such as the small example labeled "Effect of Assignment" on page 395. Extending LVN to expressions of arbitrary arity is simple.

$a + 0 = a$	$a - 0 = a$	$a - a = 0$	$2 \times a = a + a$
$a \times 1 = a$	$a \times 0 = 0$	$a \div 1 = a$	$a \div a = 1,\ a \neq 0$
$a^1 = a$	$a^2 = a \times a$	$a \gg 0 = a$	$a \ll 0 = a$
$a \text{ AND } a = a$	$a \text{ OR } a = a$	$\text{MAX}(a,a) = a$	$\text{MIN}(a,a) = a$

■ **FIGURE 8.3** Algebraic Identities for Value Numbering.

$a^2 \leftarrow b^0 + c^1$

$b^4 \leftarrow a^2 - d^3$

$c^5 \leftarrow b^4 + c^1$

$d^4 \leftarrow a^2 - d^3$

Example with Value Numbers

To see how LVN computes value numbers, consider our example, repeated in the margin. Each name is annotated with the value number that LVN assign to it, shown as a superscript. In the first operation, with an empty value table, b and c get new value numbers, 0 and 1, respectively. LVN constructs the textual string "$0 + 1$" as a hash key for the expression b + c and performs a lookup. It does not find an entry for that key, so the lookup fails. Accordingly, LVN creates a new entry for "$0 + 1$" and assigns it value number 2. LVN then creates an entry for a and assigns it the value number of the expression, namely 2. Repeating this process for each operation, in sequential order, produces the rest of the value numbers shown in the margin.

$a \leftarrow b + c$

$b \leftarrow a - d$

$c \leftarrow b + c$

$d \leftarrow b$

Example After Rewrite

The value numbers reveal, correctly, that the two occurrences of b + c produce different values, due to the intervening redefinition of b. They also show that the two occurrences of a - d produce the same value. In both occurrences of a - d, a has the same value number; d does, as well. LVN discovers this fact and records it by assigning b and d the same value number, namely 4. That knowledge lets LVN rewrite the fourth operation with a d ← b as shown in the margin. Subsequent passes may coalesce and eliminate the copy operation.

Extending the Algorithm

LVN provides a framework for several other local optimizations.

Commutative Operations Variants of a commutative expression, such as a × b and b × a, should receive the same value number. As LVN constructs a hash key for the right-hand side of the current operation, it can sort the operands using some convenient scheme, such as ordering them by value number. This simple action will ensure that variants of a commutative expression receive the same value number.

Constant Folding If all the operands of an operation have known constant values, LVN can perform the operation at compile time and fold the answer directly into the code. LVN can mark hash table entries as constants and store their values alongside their value numbers. When LVN discovers a constant expression, it can evaluate the expression and replace the expression with a literal constant value.

```
for $i \leftarrow 1$ to $n$, where the block has $n$ operations in the form "$T_i \leftarrow L_i\ Op_i\ R_i$" do
    get the value numbers $V_1$ and $V_2$ for $L_i$ and $R_i$
    if $T_i$ has no table entry then
        create a table entry for $T_i$
    if $Op_i$ commutes and $V_1 > V_2$ then
        swap $V_1$ and $V_2$
    construct a hash key, $k$, as $\langle V_1, Op_i, V_2 \rangle$
    if $k$ is already present in the table then
        replace operation $i$ with "$T_i \leftarrow x$", where $x$ is the constant value
            or name stored under the key $k$
    else
        create a table entry for $k$
        generate a new value number for $k$ and store it in $k$'s table entry
        store the name $T_i$ in $k$'s table entry
        if $V_1$ and $V_2$ are both marked as constant then
            evaluate $L_i\ Op_i\ R_i$ into $c$
            replace operation $i$ with "$T_i \leftarrow c$"
            store $c$ in the table entries for $k$ and $T_i$
        else if $L_i\ Op_i\ R_i$ matches an algebraic identity then
            replace operation $i$ with the simplified version
    store $k$'s value number in the table entry for $T_i$
```

■ **FIGURE 8.4** Local Value Numbering with Extensions.

Algebraic Identities LVN can use algebraic identities to simplify the code. For example, x + 0 and x should receive the same value number. Unfortunately, LVN needs special-case code for each identity. A series of tests, one per identity, would significantly slow down LVN. To ameliorate this problem, LVN should organize the tests into operator-specific decision trees. Since each operator has just a few identities, this approach keeps the overhead low. Fig. 8.3 shows some of the identities that can be handled in LVN.

A clever implementor will discover other identities, including some that are type specific. The exclusive-or of two identical values should yield a zero of the appropriate type. Numbers in IEEE floating-point format have their own special cases introduced by the explicit representations of ∞ and NaN; for example, $\infty - \infty = NaN$; $\infty - NaN = NaN$; and $\infty \div NaN = NaN$.

NaN

Not a Number is a defined constant that represents an invalid or meaningless result in the IEEE standard for floating-point arithmetic.

Fig. 8.4 shows LVN with these extensions. To handle commutative operations, it sorts the operands by their value numbers before it constructs the hash key. It evaluates constant expressions and applies algebraic identities.

Even with these extensions, the cost per IR operation remains extremely low because each step has an efficient implementation.

The Role of Naming

$\mathtt{a}^2 \leftarrow \mathtt{x}^0 + \mathtt{y}^1$
$\mathtt{b}^2 \leftarrow \mathtt{x}^0 + \mathtt{y}^1$
$\mathtt{a}^3 \leftarrow 17^3$
$\mathtt{c}^2 \leftarrow \mathtt{x}^0 + \mathtt{y}^1$

Example with Value Numbers

The choice of names for variables and values can limit the effectiveness of value numbering. Consider what happens when LVN is applied to the block shown in the margin. Again, the superscripts indicate the value numbers assigned to each name and value.

In the first operation, LVN assigns 0 to x, 1 to y, and 2 to both x + y and a. At the second operation, it discovers that x + y is redundant, with value number 2; thus, LVN can rewrite b ← x + y with b ← a. The third operation is not redundant. At the fourth operation, LVN again discovers that x + y has value number 2. It cannot, however, rewrite the operation as c ← a because a no longer holds value number 2.

LVN can address this problem in two distinct ways. It can keep a map from value numbers to names. At each operation, it must update the map. Say the operation defines t. LVN must remove t from the list for its old value number and add t to the list for its new value number. At a replacement, LVN can use any name currently in the map for that value number. This approach adds some cost to the processing of each operation. It also clutters up the basic algorithm.

$\mathtt{a}_0^2 \leftarrow \mathtt{x}_0^0 + \mathtt{y}_0^1$
$\mathtt{b}_0^2 \leftarrow \mathtt{x}_0^0 + \mathtt{y}_0^1$
$\mathtt{a}_1^3 \leftarrow 17^3$
$\mathtt{c}_0^2 \leftarrow \mathtt{x}_0^0 + \mathtt{y}_0^1$

Example After Renaming

As an alternative, the compiler can rewrite the code so that each operation defines a unique name. Adding a subscript to each name for uniqueness, as shown in the margin, is sufficient. With these new names, the code defines each value exactly once. Thus, no value is ever redefined and lost, or *killed*. If we apply LVN to this block, it produces the desired result. It proves that the second and fourth operations are redundant; each can be replaced with a copy from $\mathtt{a}_0$. (Notice that, after renaming, $\mathtt{a}_0$ is live over a longer region in the code than a was.)

However, the compiler must now reconcile these subscripted names with the names in surrounding blocks to preserve the meaning of the original code. In our example, the original name a should refer to the value from the subscripted name $\mathtt{a}_1$ in the rewritten code. A clever implementation would map the new $\mathtt{a}_1$ to the original a, $\mathtt{b}_0$ to the original b, and $\mathtt{c}_0$ to the original c; it would also rename $\mathtt{a}_0$ to a new temporary name. That solution reconciles the name space of the transformed block with the surrounding context without introducing copies.

This naming scheme approximates one property of the name space created for static single-assignment form, or SSA (see Sections 4.6.2 and 9.3). From

RUNTIME EXCEPTIONS AND OPTIMIZATION

Some abnormal runtime conditions can raise exceptions. Examples include out-of-bounds memory references, undefined arithmetic operations such as division by zero, and illegal operations. (One way for a debugger to trigger a breakpoint is to replace the instruction with an illegal one and to catch the exception.) Some languages include features for handling exceptions, for both predefined and programmer-defined situations.

Typically, a runtime exception causes a transfer of control to an "exception handler," code designed to deal with the exception. The handler may cure the problem, reexecute the offending operation, and return control to the block. Alternatively, it may transfer control elsewhere or terminate execution.

The optimizer must understand which operations can raise an exception and must consider the impact of an exception on program execution. Because an exception handler might modify the values of variables or transfer control, the compiler must treat exception-raising operations conservatively. For example, every exception-raising operation might end the current basic block. Such treatment can severely limit the optimizer's ability to improve the code.

To optimize exception-laden code, the compiler needs to understand and model the effects of exception handlers. To do so, it needs access to the code for the exception handlers and it needs to understand which handlers might be in place when a specific exception-raising operation executes.

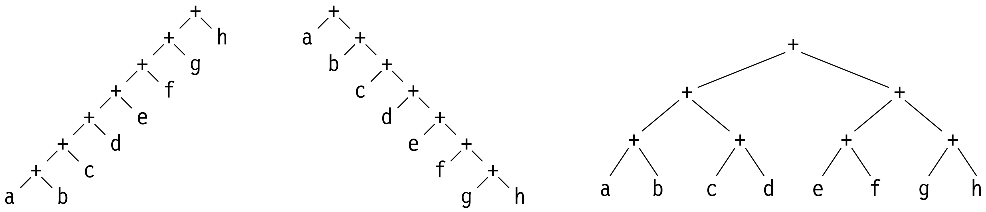
a design perspective, the simple solution is to apply LVN to code that is already in SSA form.

The Impact of Indirect Assignments

The previous discussion assumes that assignments are direct and obvious, as in a ← b × c. Many programs contain indirect assignments, where the compiler may not know which values are modified because the reference is ambiguous. Examples include (1) assignment through a pointer, such as `*p = 0;` in C; (2) assignment to a structure element or an array element, such as `a(i,j) = 0` in FORTRAN; or (3) assignment through a call-by-reference parameter. Indirect assignments complicate value numbering and other optimizations because they create imprecisions in the compiler's understanding of the flow of values.

Ambiguous reference
A reference is *ambiguous* if the compiler cannot isolate it to a single memory location.

Assume, without loss of generality, that the names in our examples are memory addresses. Consider the impact that an indirect assignment, such as `*p = 0`, must have on the state of LVN. Unless the compiler knows the address contained in p, it must assume that any location that p *might* address



(a) Left-Associative Tree (b) Right-Associative Tree (c) Balanced Tree

■ **FIGURE 8.5** Potential Tree Shapes for `a + b + c + d + e + f + g + h`.

has been changed. In an implementation of LVN where names represent memory locations, an ambiguous assignment would give each name its own new value number.

While this sounds drastic, it shows the true impact of an ambiguous assignment on the set of facts that the compiler can derive. The compiler can try to disambiguate references. For pointer-based references, it can perform analysis to narrow the set of variables that the compiler believes a pointer can address. A variety of techniques help the compiler understand the patterns of element access in an array—again, to shrink the set of locations that it must assume might be modified by an assignment to one element. Interprocedural alias analysis can resolve some of the ambiguities caused by interactions between parameter binding and name scoping rules.

8.4.2 **Tree-Height Balancing**

Expression optimization for multiple functional units demonstrates the impact that code shape can have on the compiler's ability to improve a computation. Most processors have several functional units; those units can execute independent operations in each cycle. If the compiler can express the computation so that it presents independent operations, the scheduler and the processor may be able to execute them concurrently; if, instead, the compiler introduces unnecessary sequential constraints, the scheduler and the processor may not find enough concurrency to occupy the functional units.

```
$t_1 \leftarrow a + b$
$t_2 \leftarrow t_1 + c$
$t_3 \leftarrow t_2 + d$
$t_4 \leftarrow t_3 + e$
$t_5 \leftarrow t_4 + f$
$t_6 \leftarrow t_5 + g$
$t_7 \leftarrow t_6 + h$
```

Left-to-Right Evaluation of `a + b + c + d + e + f + g + h`

Consider the expression `a + b + c + d + e + f + g + h`. The code in the margin encodes a left-to-right evaluation, which is equivalent to a postorder walk of the left-associative tree in Fig. 8.5(a). A right-to-left evaluation of the expression would correspond to the tree in panel (b). Each tree constrains the execution order in ways that are more restrictive than the rules of addition. The left-associative tree forces the code to evaluate `a + b` before it can add either `g` or `h`. The right-associative tree requires that `g + h` precede additions

	Adder 0	Adder 1
1	$t_1 \leftarrow a+b$	—
2	$t_2 \leftarrow t_1+c$	—
3	$t_3 \leftarrow t_2+d$	—
4	$t_4 \leftarrow t_3+e$	—
5	$t_5 \leftarrow t_4+f$	—
6	$t_6 \leftarrow t_5+g$	—
7	$t_7 \leftarrow t_6+h$	—

(a) Left-Associative Tree

	Adder 0	Adder 1
1	$t_1 \leftarrow a+b$	$t_2 \leftarrow c+d$
2	$t_3 \leftarrow e+f$	$t_4 \leftarrow g+h$
3	$t_5 \leftarrow t_1+t_2$	$t_6 \leftarrow t_3+t_4$
4	$t_7 \leftarrow t_5+t_6$	—
5	—	—
6	—	—
7	—	—

(b) Balanced Tree

■ **FIGURE 8.6** Schedules from Trees in Fig. 8.5.

involving a or b. The balanced tree in panel (c) imposes fewer constraints, but it still restricts evaluation order more than addition does.

If the processor can perform multiple concurrent additions, then the balanced tree should lead to a shorter schedule for the computation. Fig. 8.6 shows the shortest possible schedules for the balanced tree and the left-associative tree on a computer with two single-cycle adders. The balanced tree can execute in four cycles, with one unit idle in the fourth cycle. By contrast, the left-associative tree takes seven cycles to execute because it serializes the additions. It leaves one adder completely idle. The right-associative tree produces similar results.

This small example suggests an important optimization: using commutativity and associativity to expose additional parallelism in expression evaluation. The rest of this section presents an algorithm that rewrites a single block to improve balance across its forest of expression trees. This transformation exposes more *instruction-level parallelism* (ILP) to the instruction scheduler.

Instruction-level parallelism
Operations that are independent can execute in the same cycle. This kind of concurrency is called instruction-level parallelism.

Candidate Trees

As a first step, the algorithm must identify trees that are promising candidates for rebalancing. A candidate tree, or subtree, must contain only one kind of operator, such as + or ×. That operator must be both associative and commutative to allow rearrangement of the operands. The tree should be as large as possible, to maximize ILP. If the tree is too small, the optimization will make no difference.

To ensure that the rewritten code works in its surrounding context, that code must preserve any values that are used outside of the expression. A value

```
// All operations have the form "$T_i \leftarrow L_i\ Op_i\ R_i$"
// Phase 1: build a queue, Roots, of the candidate trees
Roots ← new empty queue
for i ← 0 to n - 1 do
    Rank($T_i$) ← -1
    if $Op_i$ is commutative and associative and $T_i$ is exposed then
        mark $T_i$ as a root
        Enqueue(Roots, $T_i$, precedence of $Op_i$)

// Phase 2: remove a tree from Roots and rebalance it
while (Roots is not empty) do
    var ← Dequeue(Roots)
    Balance(var)
```

■ **FIGURE 8.7** Tree-Height Balancing Algorithm.

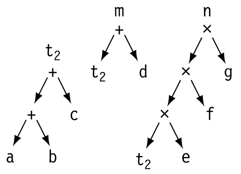
is *exposed* if it is used after the block, if it is used more than once within the block, or if it appears as an argument to an operator of another type. Exposed values must be preserved—that is, the rewritten computation must produce the same result for the exposed value as the original computation. This restriction can constrain the algorithm's ability to rearrange code.

```
$t_1$ ← a + b
$t_2$ ← $t_1$ + c
m ← $t_2$ + d
$t_3$ ← $t_2$ × e
$t_4$ ← $t_3$ × f
n ← $t_4$ × g
```

Example Basic Block

The code fragment shown in the margin computes two large expressions: m ← (a + b + c + d), and n ← (a + b + c) × e × f × g. Assume that m and n are used later, so they are exposed values. t_2 is also exposed; it must be preserved for its use in the computation of t_3.

The fact that t_2 is exposed affects the code in two distinct ways. First, the compiler cannot rewrite (a + b + c + d) as (a + b) + (c + d) because the latter expression never computes the value of t_2. Second, the computation of t_2 must precede the computation of t_3, which, again, restricts the order of operations.



Trees in Example

The algorithm to find candidate trees must treat each exposed value as the root of a tree. In the example, this produces the forest shown in the margin. Notice that t_2 is defined once and used twice. The trees for t_2 and m are too small to benefit from tree-height balancing, while the tree for n, with four operands, can be balanced.

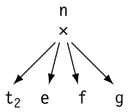
If the compiler's definitive IR is not a collection of expression trees, it can either build the expression trees for the block as a derivative IR, or it can interpret the definitive IR as expression trees to drive the algorithm. The latter approach requires some simple data structures to link each use in the block with either the definition in the block that reaches it or a notation to indicate that the use is *upward exposed*.

Upward exposed
A name *x* is *upward exposed* in block *b* if the first use of *x* in *b* refers to a value computed before entering *b*.

High-Level Sketch of the Algorithm

The tree-height balancing algorithm, shown in Fig. 8.7, consists of two phases. The first phase finds the roots of the candidate expression trees in the block. It ignores operators unless they are both commutative and associative. If a value *T* is both an exposed value and the result of a commutative and associative operator, then it adds *T* to a queue of roots, ordered by the operator's arithmetic precedence.

The second phase takes the roots, in precedence order from lowest to highest, and creates a balanced tree by flattening the tree into a single *n*-ary operator and then rebuilding it in balanced form. The *n*-ary tree for `n` in our continuing example appears in the margin.



n-ary Tree for n

Phase 1: Finding Candidate Trees

Phase 1 of the algorithm is straightforward. The algorithm walks over the operations in the block, in order, and tests each operation to see if it should be the root of a candidate tree. The test relies on observations made earlier. A candidate tree root must be the result of using a commutative and associative operator to create an exposed value.

To determine if the value T_i defined by operation *i* is exposed, the compiler must understand where T_i is used. The compiler can compute LiveOut sets for each block and use that information to determine if T_i is used after the current block (see Section 8.6.1). As part of computing the initial information for that computation, the compiler can also determine how many times T_i is used within the block.

The test for a candidate-tree root, then, can be expressed concisely. For an operation $T \leftarrow L \; Op \; R$ in block *B*, *T* is a root if and only if:

> *Op is both commutative and associative and*
> *(T is used more than once in B or T ∈ LiveOut(B))*

Phase 1 initializes the ranks of each T_i defined in the block. For each root that it finds, it marks the name as a root and adds it to a priority queue of the roots. The queue for the continuing example appears in the margin; it assumes precedence of one for + and two for ×.

(⟨t_2,1⟩, ⟨m,1⟩, ⟨n,2⟩)

Queue of Roots for Example

Phase 2: Rebuilding the Block in Balanced Form

Phase 2 takes the queue of candidate-tree roots and builds, from each root, an approximately balanced tree. Phase 2 starts with a while loop that calls *Balance* on each candidate tree root. *Balance*, *Flatten*, and *Rebuild* implement Phase 2, as shown in Figs. 8.8 and 8.9.

```
Balance(root)      // Create balanced tree from its root, $T_i$
    if Rank(root) $\geq$ 0 then
        return                              // already processed this tree

    q $\leftarrow$ new empty queue           // Flatten the tree, then rebuild it
    Rank(root) $\leftarrow$ Flatten($L_i$,q) + Flatten($R_i$,q)
    Rebuild(root, q, $Op_i$)


Flatten(node,q)    // Flatten computes a rank for node & builds the queue
    if node is a constant then              // Constant is a leaf in the tree
        Rank(node) $\leftarrow$ 0
        Enqueue(q,node,Rank(node))

    else if node is upward exposed then     // UE is a leaf in the tree
        Rank(node) $\leftarrow$ 1
        Enqueue(q,node,Rank(node))

    else if node is a root                  // Root of another tree is a leaf
        Balance(node)                       // Balance sets Rank(node)
        Enqueue(q,node,Rank(node))

    else                                    // interior node for $T_i \leftarrow L_i\ Op_i\ R_i$
        Rank(node) $\leftarrow$ Flatten($L_j$,q) + Flatten($R_j$,q)

    return Rank(node)
```

■ **FIGURE 8.8** Tree-Height Balancing Algorithm: Balance and Flatten.

Balance is invoked on each candidate-tree root in increasing priority order. If the tree has not already been balanced, *Balance* calls *Flatten* to create a new priority queue that holds all the operands of the current tree. In essence, the queue represents the single-level, *n*-ary version of the candidate tree. Once *Flatten* has created the queue, *Balance* invokes *Rebuild* to emit code for the approximately balanced tree.

Recall that Phase 1 ranked the *Roots* queue by arithmetic precedence, which forces *Flatten* to follow the correct evaluation order.

Flatten is straightforward. It recursively walks the candidate tree, assigns each node a rank, and adds it to the queue. *Flatten* assigns ranks directly to leaf nodes: zero for a constant and one for a leaf. If *Flatten* encounters a root, it calls *Balance* on that root; *Balance* creates a new queue, flattens the root's candidate tree, and rebuilds it. The root keeps the priority assigned to it by that recursive call.

When *Balance* calls *Rebuild*, the queue contains all of the leaves from the candidate tree, in rank order. Intermediate results, represented by interior nodes in the original expression trees are gone.

Balance, in effect, gathers the constants together so that *Rebuild* can combine them.

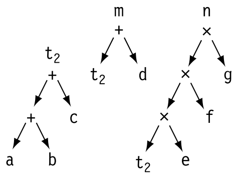
Rebuild uses a simple algorithm to rewrite the code. It repeatedly removes the two lowest-ranked items from the tree. If they are constant, then it evaluates

```
Rebuild(root, q, op)   // Build a balanced expression
    while (q is not empty) do
        NewL ← Dequeue(q)     // Get a left operand
        NewR ← Dequeue(q)     // Get a right operand

        if NewL and NewR are both constants then
            // evaluate and fold the result
            NewT ← Fold(op, NewL, NewR)
            if q is empty then
                create the op "root ← NewT"
                Rank(root) = 0
            else
                Enqueue(q, NewT, 0)
                Rank(NewT) = 0

        else if q is empty then     // not a constant; name the result
                NewT ← root
            else
                NewT ← new name

            create the op "NewT ← NewL op NewR"
            Rank(NewT) ← Rank(NewL) + Rank(NewR)

            if q is not empty then// More ops in q ⇒ add NewT to q
                Enqueue(q, NewT, Rank(NewT))
```

■ **FIGURE 8.9** Tree-Height Balancing Algorithm: Rebuild.

the operation and pushes the result back onto the queue with priority zero. Since all constants have rank zero, *Rebuild* will fold them all into a single constant term. If one or more of the items has nonzero rank, *Rebuild* creates an operation to combine the two items. It assigns the operation a new rank. Finally, it pushes the new operation onto the queue. This process continues until the queue is empty.

The expression trees and *Roots* queue that Phase 1 produces for the ongoing example are shown in the margin. Phase 2 pulls each root from the queue and passes it to *Balance*. Neither m nor t_2 have enough operands for the algorithm to create a balanced tree.

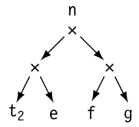


Expression Trees

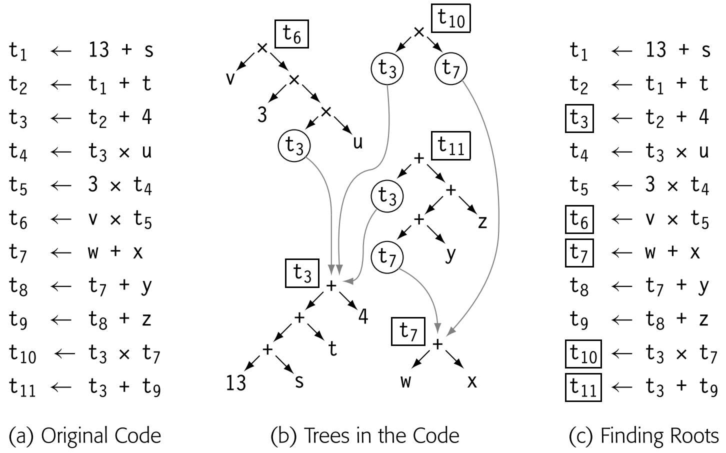
(⟨m,1⟩, ⟨t_2,1⟩,⟨n,2⟩)

Roots Queue

When Phase 2 pulls n from the *Roots* queue, *Flatten* builds a queue that holds all four operands, each with rank one. *Rebuild* then creates operations to compute g × f and t_2 × e, enters those operations into the queue with rank two, and then pulls them and combines them into the root, n. The rebuilt tree for n appears in the margin.



Rebuilt Tree for n



```
t1  ← 13 + s
t2  ← t1 + t
t3  ← t2 + 4
t4  ← t3 × u
t5  ← 3 × t4
t6  ← v × t5
t7  ← w + x
t8  ← t7 + y
t9  ← t8 + z
t10 ← t3 × t7
t11 ← t3 + t9
```

(a) Original Code

(b) Trees in the Code

```
t1  ← 13 + s
t2  ← t1 + t
t3  ← t2 + 4
t4  ← t3 × u
t5  ← 3 × t4
t6  ← v × t5
t7  ← w + x
t8  ← t7 + y
t9  ← t8 + z
t10 ← t3 × t7
t11 ← t3 + t9
```

(c) Finding Roots

■ **FIGURE 8.10** Detailed Example of Tree-Height Balancing.

A Larger Example

Fig. 8.10(a) shows a second example. Constants have been folded and redundant computations eliminated, as LVN might do. All of s, t, u, v, w, x, y, and z are upward exposed; t_6, t_{10}, and t_{11} are live on exit from the block.

The block contains several intertwined computations. Panel (b) shows the five distinct expression trees in the block. The values of t_3 and t_7 are each used multiple times, so we show them as distinct trees. The connections to t_3 and t_7 are shown in gray.

When we apply Phase 1 of the tree-height balancing algorithm to the example, it finds five roots: the three values that are live on exit, plus t_3 and t_7. The roots appear in boxes in panels (b) and (c). At the end of Phase 1, *Roots* contains: $\{\ \langle t_{11},1\rangle, \langle t_7,1\rangle, \langle t_3,1\rangle, \langle t_{10},2\rangle, \langle t_6,2\rangle\ \}$.

Phase 2 of the algorithm starts with the *Roots* queue. It removes a node from the *Roots* queue and calls *Balance* to process it. When *Balance* returns, it repeats the process until the *Roots* queue is empty.

Applying the Phase 2 algorithm to our example's *Roots* queue, it first calls *Balance* on t_{11}. Balance calls *Flatten*, which builds a queue. When *Flatten* encounters t_3, which is a root, it invokes *Balance*(t_3).

When *Balance* is applied to t_3, it first calls *Flatten*, which produces the queue: $\{\langle 4,0\rangle,\langle 13,0\rangle,\langle t,1\rangle,\langle s,1\rangle\}$. *Balance* passes this queue to *Rebuild*.

Rebuild dequeues $\langle 4,0\rangle$ and $\langle 13,0\rangle$. Since both are constants, it adds them and enqueues $\langle 17,0\rangle$. Next, it dequeues $\langle 17,0\rangle$ and $\langle t,1\rangle$; creates the operation

17 + t; assigns the result to a new name, say n_0; and enqueues $\langle n_0,1\rangle$. Finally, *Rebuild* dequeues $\langle n_0,1\rangle$ and $\langle s,1\rangle$ and creates the operation $n_0 + s$. Since the queue is now empty, *Rebuild* assigns the result to the tree's root, t_3. The code is shown in the margin.

$n_0 \leftarrow 17 + t$
$t_3 \leftarrow n_0 + s$

Rebuild returns; *Balance* returns; and control is back in the call to *Flatten* on t_{11}. *Flatten* continues, until it hits t_7, and recurs on *Balance(*t_7*)*.

For t_7, *Flatten* produces the queue: $\{\langle x,1\rangle,\langle w,1\rangle\}$. *Rebuild* dequeues $\langle x,1\rangle$ and $\langle w,1\rangle$; creates the operation $x + w$; and, since the queue is empty, assigns the result to the root, t_7, as shown in the margin.

$t_7 \leftarrow x + w$

Rebuild returns; *Balance* returns; and control returns to *Flatten(*t_{11}*)*. *Flatten* returns the queue: $\{\langle z,1\rangle,\langle y,1\rangle,\langle t_7,2\rangle\langle t_3,2\rangle\}$. *Rebuild* creates the operation $z + y$ and assigns its value to n_1. Next, it creates the operation $n_1 + t_7$ and assigns its value to n_2. Finally, it creates the operation $n_2 + t_3$ and assigns its value to the tree's root, t_{11}.

$n_1 \leftarrow z + y$
$n_2 \leftarrow n_1 + t_7$
$t_{11} \leftarrow n_2 + t_3$

Rebuild returns; *Balance* returns; and control flows back to the while loop. Phase 2 calls *Balance(*t_7*)*, which finds that t_7 has been processed. It then calls *Balance(*t_3*)*, which finds that t_3 has been processed.

Next, the while loop invokes *Balance(*t_{10}*)*. *Flatten* recurs on *Balance* for both t_3 and t_7. In both cases, the tree has already been processed. At that point, *Flatten* returns the queue: $\{\langle t_7,2\rangle, \langle t_3,2\rangle\}$. *Rebuild* creates the operation $t_7 \times t_3$ and, since the queue is now empty, assigns it to the tree's root, t_{10}. *Rebuild* returns. *Balance* returns.

$t_{10} \leftarrow t_7 \times t_3$

The next, and final, iteration of the while loop invokes *Balance(*t_6*)*. *Flatten* encounters t_3, which causes it to recur with *Balance(*t_3*)*; since t_3 has been processed, the call returns immediately. *Flatten* returns the queue: $\{\langle 3,0\rangle,\langle v,1\rangle,\langle u,1\rangle,\langle t_3,2\rangle\}$. From this queue, *Rebuild* creates the operations shown in the margin. *Rebuild* returns; *Balance* returns; and the while loop terminates. Fig. 8.11 shows the final result.

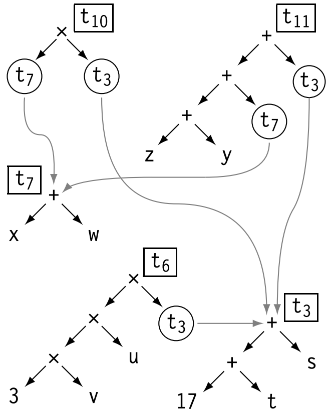
$n_3 \leftarrow 3 \times v$
$n_4 \leftarrow n_3 \times u$
$t_6 \leftarrow n_4 \times t_3$

The difference between the forests of expression trees in Figs. 8.10(b) and 8.11.b may not be obvious. To appreciate the restructured tree, compare the height of the tree for t_6, including the tree for t_3. In the original code, the tree has height seven where the restructured tree has height four. Similarly, the original tree for t_{11} has height five where the restructured tree has height four.

As a final point, the reader may observe that the individual trees in Fig. 8.11(b) do not look balanced. The algorithm creates balance across the entire block, rather than across the individual trees. This effect is a simple

```
$n_0 \leftarrow 17 + t$
$t_3 \leftarrow n_0 + s$
$t_7 \leftarrow x + w$
$n_1 \leftarrow z + y$
$n_2 \leftarrow n_1 + t_7$
$t_{11} \leftarrow n_2 + t_3$
$t_{10} \leftarrow t_7 \times t_3$
$n_3 \leftarrow 3 \times v$
$n_4 \leftarrow n_3 \times u$
$t_6 \leftarrow n_4 \times t_3$
```

(a) Transformed Code



(b) Trees in the Transformed Code

■ **FIGURE 8.11** Code Structure After Balancing.

and clear example of the difference between optimizing for a local effect and optimizing over a larger context.

The balance across the block is the result of a single line of code in *Flatten* from Fig. 8.8. For a root *x*, *Flatten* enqueues *x* with the rank it was previously assigned in *Balance*. If we modified that line of code to enqueue it with a rank of one, each of the subtrees would be balanced. The drawing might look better, but the overall height of trees that reference shared subtrees would be taller and the code would expose less ILP.

SECTION REVIEW

Local optimization operates on the code for a single basic block. These techniques rewrite the block based on information derived from that block. In the process, they must maintain the block's interactions with the surrounding execution context. In particular, they must preserve any *observable* values computed in the block.

Because they limit their scope to a single block, local optimizations can rely on properties that only hold true in straight-line code. For example, local value numbering relies on the fact that all the operations in the block execute in an order that is consistent with straight-line execution. Thus, it can build a model of prior context that exposes redundancies and constant-valued expressions. Similarly, tree-height balancing relies on the fact that a block has just one entry and one exit to determine which subexpressions in the block it must preserve and which ones it can reorder.

REVIEW QUESTIONS

1. Naming plays a key role in the effectiveness of LVN. Sketch an algorithm to rename the values in a basic block so that each definition in the block creates a unique name and that each use refers to the correct new name.
2. The tree-height balancing algorithm ranks a node n in the final expression tree with the number of nonconstant leaves below it in the final tree. How would you modify the algorithm to produce ranks that correspond to the height of n in the tree? Would that change the code that the algorithm produces?

8.5 REGIONAL OPTIMIZATION

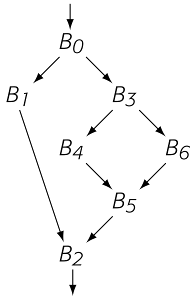
Inefficiencies are not limited to single blocks. Code in one block may provide the context for improving code in another block. Thus, many optimizations examine contexts larger than a single block.

Regional techniques operate over subsets of the control-flow graph that include multiple blocks but do not, typically, extend to the entire procedure. The primary complication that arises in the shift from local optimization to these regional methods is the need to handle multiple control-flow paths. An `if–then–else` construct introduces conditionally executed paths. Loops introduce cyclic control flow. Because regional methods examine more context than local methods, they can discover opportunites that local methods cannot see. Because they focus on a subset of the procedure's CFG, these methods can often be simpler than a global approach to the same problem.

This section presents two regional techniques as illustrations. The first, superlocal value numbering, extends LVN to multiple blocks. The second, loop unrolling, operates over a single loop nest; unrolling played a role in the discussion of `dmxpy` in Section 8.2.1.

8.5.1 Superlocal Value Numbering

To improve the results of LVN, the compiler can expand its scope from a single basic block to an extended basic block, or EBB (see Section 8.3). To process an EBB, the algorithm should value number each path through the EBB. Consider, for example, the CFG shown in Fig. 8.12(a). It contains three EBBs. The nontrivial EBB is $(B_0, B_1, B_3, B_4, B_6)$. The other two EBBs are trivial: (B_2) and (B_5). The regional version of value numbering, which we call *superlocal value numbering* (SVN), processes one EBB at a time and propagates information down each path in the EBB.



(a) Control-Flow Graph

```
$B_0$: $m_0 \leftarrow a_0 + b_0$
     $n_0 \leftarrow a_0 + b_0$
     $(a_0 > b_0) \rightarrow B_1, B_3$

$B_1$: $p_0 \leftarrow c_0 + d_0$
     $r_0 \leftarrow c_0 + d_0$
     $\rightarrow B_2$

$B_2$: $r_2 \leftarrow \phi(r_0, r_1)$
     $y_0 \leftarrow a_0 + b_0$
     $z_0 \leftarrow c_0 + d_0$
     $\rightarrow$ exit

$B_3$: $q_0 \leftarrow a_0 + b_0$
     $r_1 \leftarrow c_0 + d_0$
     $(a_0 > b_0) \rightarrow B_4, B_6$

$B_4$: $e_0 \leftarrow b_0 + 18$
     $s_0 \leftarrow a_0 + b_0$
     $u_0 \leftarrow e_0 + f_0$
     $\rightarrow B_5$

$B_5$: $e_2 \leftarrow \phi(e_0, e_1)$
     $u_2 \leftarrow \phi(u_0, u_1)$
     $v_0 \leftarrow a_0 + b_0$
     $w_0 \leftarrow c_0 + d_0$
     $x_0 \leftarrow e_2 + f_0$
     $\rightarrow B_2$

$B_6$: $e_1 \leftarrow a_0 + 17$
     $t_0 \leftarrow c_0 + d_0$
     $u_1 \leftarrow e_1 + f_0$
     $\rightarrow B_5$
```

(b) Corresponding Low-Level Code

■ **FIGURE 8.12** Example for Superlocal Value Numbering.

The nontrivial EBB contains three paths: $\langle B_0, B_1\rangle$, $\langle B_0, B_3, B_4\rangle$, and $\langle B_0, B_3, B_6\rangle$. SVN treats each of these paths as if it were straight-line code. To process $\langle B_0, B_1\rangle$, the compiler can apply LVN to B_0 and use the resulting hash table as a starting point when it applies LVN to B_1. The same approach can handle $\langle B_0, B_3, B_4\rangle$ and $\langle B_0, B_3, B_6\rangle$ by processing the blocks for each in order and carrying the hash tables forward. Blocks that have multiple predecessors, such as B_2 and B_5, start new EBBs and, therefore, inherit no context from their predecessors.

This scheme achieves the results of LVN over longer paths by treating each EBB path as if it were a single block. To make the example concrete, consider the code for each block shown in Fig. 8.12(b). Applying LVN to EBB paths exposes opportunities that are hidden when considering just the individual blocks.

- In $\langle B_0, B_1\rangle$, LVN discovers that the assignments to n_0 and r_0 are redundant. SVN discovers the same redundancies.
- In $\langle B_0, B_3, B_4\rangle$, LVN finds that the assignment to n_0 is redundant. SVN also finds that the assignments to q_0 and s_0 are redundant.
- In $\langle B_0, B_3, B_6\rangle$, LVN finds that the assignment to n_0 is redundant. SVN also finds that the assignments to q_0 and t_0 are redundant.
- In B_5 and B_2, SVN degenerates to LVN.

The difficulty in this approach lies in making the process efficient. The obvious approach would treat each path as if it were a single block, pretending,

```
// Start the process
WorkList ← { entry block }
Empty ← new table for Block
while (WorkList is not empty) do
    remove Block from WorkList
    SVN(Block, Empty)
free Empty
```

```
// Superlocal value numbering algorithm
SVN(Block, Table)
    t ← new table for Block
    link Table as the surrounding scope for t
    LVN(Block, t)
    for each successor s of Block do
        if |preds(s)| = 1 then
            SVN(s, t)
        else if s has not been processed then
            add s to WorkList
    free t
```

■ **FIGURE 8.13** Superlocal Value Numbering Algorithm.

for example, that the code for $\langle B_0, B_3, B_4 \rangle$ looks like the code shown in margin. Unfortunately, this approach analyzes a block once for each path that includes it. In the extended block $(B_0, B_1\ B_3\ B_4, B_6)$, the algorithm would analyze B_0 three times and B_3 twice. While we want the optimization benefits that come from examining increased context, we also want to minimize compile-time costs. The key to achieving these conflicting goals is to capitalize on the tree structure of the EBB.

```
m0 ← a0 + b0
n0 ← a0 + b0
q0 ← a0 + b0
r1 ← c0 + d0
e0 ← b0 + 18
s0 ← a0 + b0
u0 ← e0 + f0
```

Treating a Path as a Single Block

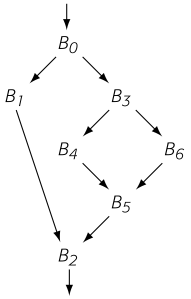
To make SVN efficient, the compiler must reuse the results of blocks that occur as prefixes on multiple paths through the EBB. After processing $\langle B_0, B_3, B_4 \rangle$, it needs a mechanism to recreate the state for the end of $\langle B_0, B_3 \rangle$ so that it can reuse that state to process B_6. The compiler can accomplish this effect in several ways:

- It can record the state of the table at each block boundary and restore that state when needed.
- It can unwind the effects of a block by walking the block backward and, at each operation, undoing the work of the forward pass.
- It can use a sheaf of linked hash tables to implement the value table (see Section 4.5.1). As it enters a block, it creates a new table. To retract the block's effects, it deletes that block's table.

While all three schemes will work, the sheaf of tables can produce a simple and fast implementation, particularly if the compiler writer can reuse a table implementation from the compiler's front end.

SVN can easily estimate the size of each table from the block sizes.

Fig. 8.13 shows a high-level sketch of the SVN algorithm, using a sheaf of value tables. It assumes that the LVN algorithm has been parameterized to accept a block and a sheaf of tables. At each block *b*, SVN allocates a value table for *b*, links it to the value table of the predecessor block as if



Example's CFG

it were a surrounding scope, and invokes LVN on block b with this new table. When LVN returns, SVN must decide how to process each of b's successors.

For a successor s of b, two cases arise. If s has only one predecessor, b, then SVN should use the accumulated context from b. Thus, SVN recurs on s with the table from b. If s has multiple predecessors, then SVN must start s with a new context, so SVN adds s to the *WorkList*. The outer loop will find it later and invoke SVN on s with an empty table. It is important to implement *Worklist* as a set to ensure that each block is processed exactly once (see Section B.2).

One complication remains. A name's value number is recorded in the value table associated with the first operation in the EBB that defines it. This effect can defeat our use of the scoping mechanism. In our example CFG, if a name x were defined in each of B_0, B_4, and B_6, its value number would be recorded in the scoped table for B_0. When SVN processed B_4, it would record x's new value number from B_4 in the table for B_0. When SVN deleted the table for B_4 and created a new table for B_6, the value number from the definition in B_4 would remain.

To avoid this complication, the compiler can run SVN on a representation that defines each name once. As we saw in Section 4.6.2, SSA form has the requisite property; each name is defined at exactly one point in the code. Using SSA form ensures that SVN records the value number for a definition in the table that corresponds to the block that contains the definition. With SSA form, SVN can undo all of a block's effects by deleting its value table. This action reverts the set of tables to their state on exit from the block's CFG predecessor. As discussed in Section 8.4.1, using SSA form can also make LVN more effective.

1. *Create scope for* B_0
2. *Apply* LVN *to* B_0
3. *Create scope for* B_1
4. *Apply* LVN *to* B_1
5. *Add* B_2 *to WorkList*
6. *Delete* B_1*'s scope*
7. *Create scope for* B_3
8. *Apply* LVN *to* B_3
9. *Create scope for* B_4
10. *Apply* LVN *to* B_4
11. *Add* B_5 *to WorkList*
12. *Delete* B_4*'s scope*
13. *Create scope for* B_6
14. *Apply* LVN *to* B_6
15. *Delete* B_6*'s scope*
16. *Delete* B_3*'s scope*
17. *Delete* B_0*'s scope*
18. *Create scope for* B_5
19. *Apply* LVN *to* B_5
20. *Delete* B_5*'s scope*
21. *Create scope for* B_2
22. *Apply* LVN *to* B_2
23. *Delete* B_2*'s scope*

Actions Taken by SVN
on the Example

Applying the algorithm from Fig. 8.13 to the code shown in Fig. 8.12 produces the sequence of actions shown in the margin. The algorithm begins with B_0 and proceeds down to B_1. At the end of B_1, it visits B_2, realizes that B_2 has multiple predecessors, and adds it to the worklist. Next, it backs up and processes B_3 and then B_4. At the end of B_4, it adds B_5 to the worklist. It then backs up to B_3 and processes B_6. At that point, control returns to the while loop, which invokes SVN on the two singleton blocks from the worklist, B_5 and B_2.

SVN discovers and removes redundant computations that LVN cannot. In the example, it finds that the assignments to $\texttt{q}_0$, $\texttt{s}_0$, and $\texttt{t}_0$ are redundant because of definitions in earlier blocks. LVN, with its purely local scope, cannot find these redundancies.

```
   do 60 j = 1, n2
      nextra = mod(n1,4)
      if (nextra .ge. 1) then
         do 49 i = 1, nextra
            y(i) = y(i) + x(j) * m(i,j)
49          continue

      do 50 i = nextra + 1, n1, 4
         y(i)   = y(i)   + x(j) * m(i,j)
         y(i+1) = y(i+1) + x(j) * m(i+1,j)
         y(i+2) = y(i+2) + x(j) * m(i+2,j)
         y(i+3) = y(i+3) + x(j) * m(i+3,j)
50       continue
60    continue
```

(a) Unroll Inner Loop by Four

```
   nextra = mod(n2,4)
   if (nextra .ge. 1) then
      do 59 j = 1, nextra
         do 49 i = 1, n1
            y(i) = y(i) + x(j) * m(i,j)
49          continue
59       continue

   do 60 j = nextra+1, n2, 4
      do 50 i = 1, n1
         y(i) = y(i) + x(j)   * m(i,j)
         y(i) = y(i) + x(j+1) * m(i,j+1)
         y(i) = y(i) + x(j+2) * m(i,j+2)
         y(i) = y(i) + x(j+3) * m(i,j+3)
50       continue
60    continue
```

(b) Unroll Outer Loop by Four, Fuse Inner Loops

■ **FIGURE 8.14** Unrolling dmxpy's Loop Nest.

On the other hand, SVN has its own limitations. It fails to find redundancies in B_5 and B_2. The reader can tell, by inspection, that the assignments in those blocks are redundant. Because the blocks have multiple predecessors, SVN cannot carry context into them. Thus, it misses those opportunities. An algorithm would need more context to discover them.

8.5.2 **Loop Unrolling**

Loop unrolling is, perhaps, the oldest, simplest, and best-known loop transformation. To unroll a loop, the compiler replicates the loop's body and adjusts the logic that controls the number of iterations performed. To see this, consider the loop nest from dmxpy used as an example in Section 8.2.

```
   do 60 j = 1, n2
      do 50 i = 1, n1
         y(i) = y(i) + x(j) * m(i,j)
50    continue
60 continue
```

The compiler can unroll either the inner loop or the outer loop. The result of unrolling the inner loop by four is shown in Fig. 8.14(a). Unrolling the outer loop by four produces four inner loops; if the compiler then combines those inner-loop bodies—a transformation called *loop fusion*—it will produce code similar to that shown in Fig. 8.14(b). The combination of outer-loop unrolling followed by inner-loop fusion is often called *unroll-and-jam*.

Loop fusion
The process of combining two loop bodies into one is called *fusion*.

Fusion is safe when each definition and each use in the resulting loop has the same value that it did in the original loops.

In each case, the transformed code needs a short prolog loop to peel off enough iterations so that the unrolled loop processes an integral multiple of four iterations. If the loop bounds are known at compile time, the compiler can determine if the prolog is necessary.

Access to m is sequential because FORTRAN stores arrays in column-major order.

Inner-loop unrolling and outer-loop unrolling produce different results for this particular loop nest. Inner-loop unrolling reduces the number of test-and-branch sequences that the inner loop executes. By contrast, outer-loop unrolling followed by inner-loop fusion not only reduces the number of test-and-branch sequences, but also produces reuse of y(i) and sequential access to both x and m. The increased reuse fundamentally changes the ratio of arithmetic operations to memory operations in the loop; undoubtedly, the author of dmxpy had that effect in mind when he hand-optimized the code, as shown in Fig. 8.1. As discussed below, each approach may also accrue indirect benefits.

Sources of Improvement and Degradation

Loop unrolling has both direct and indirect effects on the code that the compiler can produce for a given loop. The final performance of the loop depends on all of the effects, direct and indirect.

In terms of direct benefits, unrolling should reduce the number of operations required to complete the loop. The control-flow changes reduce the total number of test-and-branch sequences. Unrolling can create reuse within the loop body, reducing memory traffic. Finally, if the loop contains a cyclic chain of copy operations, unrolling can eliminate the copies as shown in Exercise 8.6.

Unrolling a loop can, however, increase the code size, both in the IR form and in the final form as executable code. Growth in IR increases compile time; growth in executable code has little effect until the loop overflows the instruction cache—at which time the degradation probably overwhelms any direct benefits.

The compiler can also unroll for indirect effects. A key side effect of unrolling is to increase the number of operations in the loop body. Other optimizations can capitalize on this change in several ways:

- Increasing the amount of ILP in the loop body can lead to better instruction schedules. With more independent operations, the scheduler has a better chance of keeping multiple functional units busy and of hiding the latency of multicycle operations such as branches and memory accesses.

- Unrolling can move consecutive memory accesses into the same iteration, where the compiler can schedule them together. That may improve locality or allow the use of multiword operations.
- Unrolling can expose cross-iteration redundancies that are harder to discover in the original code. Both versions of the code in Fig. 8.14 reuse address expressions across iterations of the original loop. In the unrolled loops, local value numbering would find and eliminate those redundancies. In the original, it would miss them.
- The unrolled loop may optimize differently than the original loop. For example, increasing the number of appearances of a variable in a loop can change the weights used in spill code selection within the register allocator (see Section 13.4). Changing the pattern of register spills can radically affect the running time of the loop.

Note, however, that the unrolled loop body may need more registers than the original loop body. If the increased demand for registers induces additional register spills and restores, then the resulting memory traffic may overwhelm the potential benefits of unrolling.

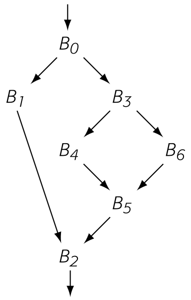
These indirect interactions are much harder to characterize and understand than the direct effects. They can produce significant performance improvements. They can also produce performance degradations. The difficulty of predicting such indirect effects has led some researchers to advocate an adaptive approach to choosing unroll factors; in such systems, the compiler tries several unroll factors and measures the performance characteristics of the resulting code.

SECTION REVIEW

Optimizations that focus on regions larger than a block and smaller than an entire procedure can find and capitalize on opportunities that a purely local algorithm cannot. These regional techniques have a long history in the literature and in practice.

In some cases, such as SVN, the extension to a regional scope provides increased optimization for little additional cost. Both SVN and superlocal instruction scheduling (see Section 12.4.1) have efficient implementations.

In other cases, regional optimizations apply more aggressive techniques to small regions in the code where the payoff may be large. Loop optimizations fit this mold. Many loop optimizations require extensive analysis to prove safety. Because so many programs spend a large part of their execution time inside loops, compiler writers have found these techniques to be profitable.



REVIEW QUESTIONS

1. Superlocal value numbering extends local value numbering to extended basic blocks through clever use of a scoped hash table. Consider the issues that might arise in extending the tree-height balancing algorithm to a superlocal scope.
   a. How would your algorithm handle a single path through an EBB, such as $\langle B_0, B_3, B_4 \rangle$ in the control-flow graph shown in the margin?
   b. What complications arise when the algorithm tries to process $\langle B_0, B_3, B_6 \rangle$ after processing $\langle B_0, B_3, B_4 \rangle$?
2. Loop unrolling always decreases loop overhead. Despite this fact, compilers do not unroll every loop. What factors should the compiler consider in deciding whether or not to unroll a given loop?

8.6 GLOBAL OPTIMIZATION

Global optimizations operate on an entire procedure or method. Because their scope includes cyclic control-flow constructs such as loops, these methods typically complete an analysis phase before they rewrite the code.

This section presents two examples of global analysis and optimization. The first, finding uninitialized variables with live information, is not strictly an optimization. Rather, it uses global data-flow analysis to discover useful information about the flow of values in a procedure. We will use the discussion to introduce the notion of *liveness*, which plays a role in many optimization techniques, including tree-height balancing (Section 8.4.2), the construction of SSA form (Section 9.3), and register allocation (Chapter 13). The second, global code placement, uses profile information gathered from running the compiled code to rearrange the layout of the executable code.

8.6.1 Finding Uninitialized Variables with Live Sets

If a procedure can use some variable v before v has been assigned a value, we say that v is potentially uninitialized at that use. Use of an uninitialized variable almost always indicates a logical error in the procedure being compiled. If the compiler discovers such situations, it should alert the programmer to their existence.

Live variable
A variable v is *live* at point p if there exists a path from p to a use of v along which v is not redefined.

We can find potential uses of uninitialized variables by computing information about *liveness*. A variable v is *live* at point p if and only if there exists a path in the CFG from p to a use of v along which v is not redefined. We encode live information by computing, for each block b in the procedure, a set LiveOut(b) that contains all the variables that are live on exit from b.

Conceptually, the compiler can find uninitialized variables by adding an artificial entry node, n_0, to the CFG and an edge from n_0 to each actual entry node in the CFG. Then, LiveOut(n_0) contains precisely those variables that can be used before they are defined.

In practice, since n_0 would have no effect on the sets of the other nodes in the CFG, the compiler need not represent n_0. It can solve for LiveOut ignoring n_0. After that computation, it can calculate the value that LiveOut(n_0) would have, if n_0 existed. This approach should be marginally faster, since it avoids iterative evaluations of LiveOut(n_0).

The computation of LiveOut sets is an example of *global data-flow analysis*, a family of techniques for reasoning, at compile time, about the flow of values at runtime. Problems in data-flow analysis are typically posed as a set of simultaneous equations over sets associated with the nodes and edges of a graph.

Data-flow analysis
a form of compile-time analysis for reasoning about the flow of values at runtime

Defining the Data-Flow Problem

Live variable analysis is a classic problem in global data-flow analysis. The compiler computes, for each node n in the procedure's CFG, a set LiveOut(n) that contains all the variables that are live on exit from the block corresponding to n.

Data-flow problems are often defined by an equation over sets associated with the nodes of the CFG. For live variables, the following equation defines the LiveOut set for a node n:

$$\textsc{LiveOut}(n) = \bigcup_{m \in succ(n)} \left(\begin{array}{l} \textsc{UEVar}(m) \cup \\ (\textsc{LiveOut}(m) \cap \overline{\textsc{VarKill}(m)}\,) \end{array} \right)$$

where $succ(n)$ refers to the set of CFG successors of n. The analysis should initialize LiveOut(n) $= \emptyset,\ \forall n$.

Here, UEVar(m) contains the upward-exposed variables in m—variables that are used in m before being defined in m. VarKill(m) contains any variables defined in m; the overline on VarKill(m) indicates its logical complement, the set of all variables not defined in m. The compiler must compute both UEVar and VarKill sets for each node in the CFG before it can solve the equations for LiveOut.

The equation encodes the definition of liveness in an intuitive way. LiveOut(n) is just the union of those variables that are live at the head

```
// each block has k operations
// of the form "x ← y op z"
for each block b do
    // Initialize the sets for b
    UEVAR(b) ← ∅
    VARKILL(b) ← ∅
    for i ← 1 to k do
        if y ∉ VARKILL(b) then
            add y to UEVAR(b)
        if z ∉ VARKILL(b) then
            add z to UEVAR(b)
        add x to VARKILL(b)
```

(a) Gathering Initial Information

```
// assume the CFG has N blocks
// numbered 0 to N - 1
for i ← 0 to N - 1 do
    LIVEOUT(i) ← ∅
changed ← true
while (changed) do
    changed ← false
    for i ← 0 to N - 1 do
        recompute LIVEOUT(i)
        if LIVEOUT(i) changed then
            changed ← true
```

(b) Solving the Equations

■ **FIGURE 8.15** Iterative Live Analysis.

of some block m that immediately follows n in the CFG. The definition requires that a value be live on some path, not on all paths. Thus, the contributions of the successors of n in the CFG are combined to form LIVEOUT(n). The contribution of a specific successor m of n is:

$$\text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})$$

A variable, v, is live on entry to m under one of two conditions. It can be used in m before it is defined in m: $v \in \text{UEVAR}(m)$. It can be live on exit from m and pass unscathed through m because m does not redefine it: $v \in \text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)}$. Combining these two terms gives the contribution of m to LIVEOUT(n). To compute LIVEOUT(n), the analyzer computes the contribution of each of n's successors, denoted $\text{m} \in succ(n)$, and combines them with a union operation.

Solving the Data-Flow Problem

To compute the LIVEOUT sets for a procedure and its CFG, the compiler can use a three-step algorithm.

1. *Build a CFG* The compiler steps through the blocks and builds the graph (see Section 4.4.4). Each node represents a block.
2. *Gather Initial Information* For each block b, the compiler computes UEVAR(b) and VARKILL(b), as shown in Fig. 8.15(a).
3. *Solve the Equations for LIVEOUT(b)* Fig. 8.15(b) shows a simple iterative fixed-point algorithm that will solve the equations.

	LiveOut(n)				
Iteration	B_0	B_1	B_2	B_3	B_4
–	∅	∅	∅	∅	∅
1	{ i }	{ s,i }	{ s,i }	{ s,i }	∅
2	{ s,i }	{ s,i }	{ s,i }	{ s,i }	∅
3	{ s,i }	{ s,i }	{ s,i }	{ s,i }	∅

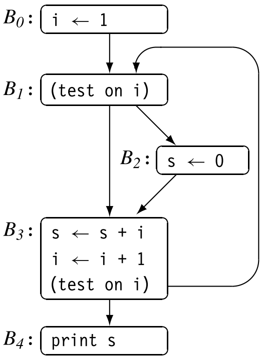
■ **FIGURE 8.16** Example LiveOut Computation.

The rest of this section works through an example LiveOut computation. Section 9.2 explores iterative data-flow analysis in more depth.

Gathering Initial Information

To compute LiveOut, the analyzer needs UEVar and VarKill sets for each block. A single pass can compute both. For each block, the analyzer initializes these sets to ∅. Next, it walks the block, in order from top to bottom, and updates both UEVar and VarKill to reflect the impact of each operation. Fig. 8.15(a) shows the details of this computation for operations of the form $x \leftarrow y \langle op \rangle$ z.

Consider the CFG shown in the margin. It consists of a simple loop that contains an `if-then` construct. The code abstracts away the details of the compares and branches. The table below the CFG shows the initial UEVar and VarKill sets.



CFG for the Example

	UEVar	VarKill
B_0	∅	{i}
B_1	{i}	∅
B_2	∅	{s}
B_3	{s,i}	{s,i}
B_4	{s}	∅

Initial Sets

Solving the LiveOut Equations

Given UEVar and VarKill sets for each node in the CFG, the compiler applies the algorithm from Fig. 8.15(b) to compute LiveOut sets for each node. It initializes all of the LiveOut sets to ∅. Next, it computes the LiveOut set for each block, in order. It repeats the process, computing LiveOut for each node in order until the LiveOut sets no longer change.

The table in Fig. 8.16 shows the values of the LiveOut sets at each iteration of the solver. Iteration zero shows the initial values. Iteration one computes an initial approximation to the LiveOut sets. Because it processes the blocks in ascending order of their labels, B_0, B_1, and B_2 receive values based solely on the UEVar sets of their CFG successors. When the algorithm reaches B_3, it has already computed an approximation for LiveOut(B_1), so the value that it computes for B_3 reflects the contribution of the new value for LiveOut(B_1). LiveOut(B_4) is empty, as befits the exit block.

In the second iteration, the value s is added to LiveOut(B_0) as a consequence of its presence in the approximation of LiveOut(B_1). No other changes occur. The third iteration does not change the values of the LiveOut sets and the algorithm halts.

The theory of iterative data-flow analysis guarantees that the solution to this set of equations is unique (see also Section 9.2).

The order in which the algorithm processes the blocks affects the values of the intermediate sets. If the algorithm visited the blocks in descending order of their labels, it would require one fewer pass. However, the final values of the LiveOut sets are independent of the evaluation order. The iterative solver in Fig. 8.15 computes a fixed-point solution to the equations for LiveOut.

This iterative data-flow solver is a fixed-point algorithm. It halts because (1) the LiveOut sets are finite and (2) the recomputation of LiveOut for a block can either produce the same set or a larger set. The VarKill and UEVar sets are constant across all the iterations. The LiveOut sets increase monotonically; since their size is bounded, the algorithm must halt.

Finding Uninitialized Variables

Once the compiler has computed LiveOut sets for each node in the CFG, it must compute the set of variables that are live on entry to any of the CFG's entry nodes. This set is equivalent to LiveOut(n_0), if n_0 were an artificial entry node with an edge to each actual entry node.

The example CFG has just one entry node, B_0, so LiveOut(n_0) is:

$$\text{UEVar}(B_0) \cup (\text{LiveOut}(B_0) \cap \overline{\text{VarKill}(B_0)}) = \{ \texttt{s} \}$$

Inspection shows that s is uninitialized along the path $\langle B_0, B_1, B_3 \rangle$.

This approach identifies variables that have a potentially uninitialized use. The compiler should recognize that situation and report it to the programmer. However, this approach may yield false positives for several reasons.

```
 ...
 p = &x;
*p = 0;
 ...
 x = x+1;
```

- If a variable *v* is ambiguous, it could be initialized through another name. Live analysis will not connect the initialization and the use. This situation can arise when a pointer is set to the address of a local variable, as in the code fragment shown in the margin.
- If a variable *v* exists before the current procedure is invoked, then it may have been previously initialized in a manner invisible to the analyzer. This case can arise with static variables of the current scope or with variables declared outside the current scope.
- The equations for live analysis may discover a path from the procedure's entry to a use of *v* along which *v* is not defined. If that path is not feasible at runtime, then *v* will still appear in LiveOut(n_0) even though no execution will ever use the uninitialized value. For example, the C program in the margin always initializes s before its use, yet s ∈ LiveOut(n_0).

```
main() {
  int s, i = 1;
  while (i<=80) {
    if (i==1)
      s = 0;
    s = s + i++;
  }
  printf("%d",s);
}
```

The marginal example with the while loop illustrates one of the fundamental limits of data-flow analysis: it assumes that all paths through the CFG are feasible at runtime. In the example, that assumption is too conservative. The only CFG path that leads to an uninitialized use runs from the entry of `main` into the loop, bypasses the initialization of `s`, and executes the increment of `s`. That path can never occur, because `i` must have the value `1` on the loop's first iteration. The LIVEOUT solver cannot discover that fact because it has no mechanism to understand that the path is infeasible.

The assumption that all paths in the CFG are feasible greatly reduces the cost of the analysis. At the same time, the assumption produces a loss of precision in the computed sets. To discover that `s` is initialized on the first iteration of the while loop, the compiler would need to combine an analysis that tracked individual paths with some form of constant propagation and with live analysis. To solve the problem in a general way would require symbolic evaluation of parts of the code during the analysis, a much more expensive prospect.

Another kind of complication arises at a procedure call. If a procedure contains a call site, the analyzer must account for any side effects of that call. At a call site s, some global variable v might be either used or modified as the result of a call to s. (Similar issues arise if v is passed as a call-by-reference parameter or if the callee is nested inside the caller.) In the absence of specific information about the callee, the analyzer must assume that every variable that might be modified is modified and that any variable that might be used is used. Such assumptions are safe, in that they represent the worst-case behavior.

Other Uses for Live Variables

Compilers use liveness in many other contexts.

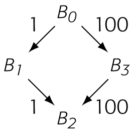
- The compiler can use live information to discover useless store operations. An operation that stores an unambiguous scalar variable v to memory is useless unless v is live at the store. A useless store can be deleted.
- Live information can improve the speed and precision of the SSA construction; a value only needs a ϕ-function in a block where it is live. Eliminating nonlive ϕ-functions can significantly reduce the number of ϕ-functions that the compiler inserts.
- Live-variable information plays a critical role in global register allocation (see Section 13.4). The allocator need not preserve values in regions where they are not live. Live information forms the basis for reasoning about which values to keep in registers.

In different contexts, liveness is calculated for different sets of names. We have discussed LiveOut with an implicit domain of variable names. In global register allocation, the compiler computes LiveOut sets over a carefully constructed set of values, called *live ranges*.

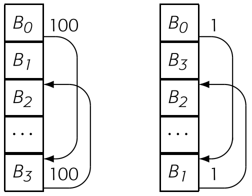
Live range
The live range of a value is the set of operations that lie between its definitions and its uses.

8.6.2 Global Code Placement

Many processors have asymmetric branch costs; the cost of the fall-through path is less than the cost of the taken branch. Each branch has two successor basic blocks; the compiler can choose which block lies on the fall-through path and which lies on the taken path. The global code-placement optimization relies, implicitly, on the twin observations that branches have asymmetric costs and that some decisions in the code have lopsided execution frequencies.



Consider the CFG shown in the margin. (B_0, B_3) executes 100 times more often than (B_0, B_1). If branch costs are asymmetric, the compiler should use the less expensive branch for (B_0, B_3). If (B_0, B_1) and (B_0, B_3) have roughly equal execution frequencies, then that choice would have little impact for this example.



Slow Layout Fast Layout

Two different layouts for this code are shown in the margin. The "slow" layout uses the fall-through branch to implement (B_0, B_1) and the taken branch for (B_0, B_3). The "fast" layout reverses this decision. If the fall-through branch is faster than the taken branch, then the "fast" layout uses the less expensive branch 100 times more often.

The compiler can take advantage of asymmetric branch costs. If the compiler knows the expected relative execution frequencies of the branches in a procedure, it can lay out the code for faster execution.

To perform global code placement, the compiler reorders the basic blocks of a procedure according to two principles:

- First, the compiler should make the most likely execution paths use fall-through branches. Thus, whenever possible, a block should be followed immediately by its most frequent successor.

  This placement has two benefits. A larger portion of executed branches take the faster (fall-through) path, directly reducing execution time. Placing consecutive blocks in consecutive virtual memory addresses should also improve instruction cache locality.
- Second, the compiler should move infrequently executed code (cold code) to the end of the procedure. This action places frequently executed code (hot code) near other hot code and cold code away from the hot code.

GATHERING PROFILE DATA

Profile data plays an important role in optimizations such as global code placement or inline substitution (see Section 8.7.1). Several approaches are used to gather profile data.

- *Instrumented Executables* The compiler can generate code to count specific events, such as procedure entries and exits or taken branches. At the end of execution, the data is written to an external file and processed offline by another tool.
- *Timer Interrupts* The profile tool can interrupt program execution at regular, frequent intervals. The tool constructs a histogram of program counter locations where the interrupts occurred. Again, postprocessing constructs a profile.
- *Performance Counters* Most processors support hardware counters that record specific events, such as taken branches. The runtime system can use counters to construct accurate profile-like data.

Each approach produces slightly different data. An instrumented executable can measure many different runtime properties. A timer-interrupt system may have lower overhead, but only finds frequently executed statements (not the paths taken to reach them). Hardware counters are accurate and efficient, but depend in idiosyncratic ways on the specific processor.

Each of these approaches works. Each of them requires cooperation between the compiler and the profiling tool on issues such as data formats, code layout, and methods for mapping runtime locations back to locations in the source code or the IR.

This placement can reduce the working-set size of the hot code, which can, in turn, improve locality in cache and virtual memory.

After placement, the code should execute longer sequences of operations without disruption—a taken branch, a stall due to an instruction cache fault, or a stall due to a page fault.

Code placement, like most global optimizations, has separate analysis and transformation phases. The analysis phase constructs branch execution frequency estimates. The transformation phase uses that data, expressed as weights on CFG edges, to discover the frequently executed paths. It then reorders the basic blocks from that model.

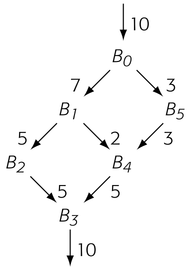
Obtaining Profile Data

For global code placement, the compiler needs estimates of the relative execution frequency of each edge in the CFG. It can obtain that information

```
E ← |edges|
for each block b do
    create a degenerate chain, d, for b
    priority(d) ← E
P ← 0
for each CFG edge ⟨x,y⟩, x ≠ y, in decreasing frequency order do
    if x is the tail of chain a and y is the head of chain b then
        t ← priority(a)
        append b onto a
        priority(a) ← min(t, priority(b), P++)
    else priority(b) ← min(priority(b), P++)
```

■ **FIGURE 8.17** Building Hot Paths.

from a profiling run of the code: compile the entire program, run it under a profiling tool on representative data, and give the compiler access to the resulting profile data. It can obtain that information from a model of program execution; such models range from simple to elaborate, with a range of accuracies.



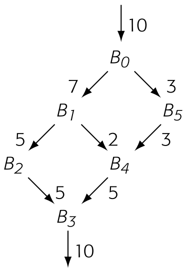
Example CFG

Specifically, the compiler needs execution counts for the CFG edges. The CFG in the margin illustrates why edge counts are superior to block counts for code placement. From the execution counts, shown as labels on the edges, we see that blocks B_0 and B_3 each execute ten times. The path $\langle B_0, B_1, B_2, B_3 \rangle$ executes more than any other path in this CFG fragment. The edge counts suggest, for example, that making B_2 the fall-through case of the branch at the end of B_1 is better than making it the taken case.

Using execution counts for blocks rather than edges can be misleading; a block's count can combine counts from multiple paths. In the CFG, the block counts for B_2 and B_4 would each be 5. For the branch at the end of B_1, those block counts suggest that the paths to B_2 and B_4 are of equal importance, while the edge counts show that (B_1, B_2) executes more often than (B_1, B_4). The code-placement algorithm ranks edges by frequency of execution; thus, the increased accuracy from the edge counts has a direct effect on the quality of the results.

Constructing Chains as Hot Paths in the CFG

To determine the code layout, the compiler finds *hot paths* through the CFG—paths that contain the most frequently executed edges. Each path is a chain of one or more blocks. The algorithm assigns a priority to each path; the priorities drive the code reordering process.



(a) Example CFG

Edge	Set of Chains	P
—	$\langle B_0\rangle_E$, $\langle B_1\rangle_E$, $\langle B_2\rangle_E$, $\langle B_3\rangle_E$, $\langle B_5\rangle_E$, $\langle B_4\rangle_E$	0
(B_0,B_1)	$\langle B_0,B_1\rangle_0$, $\langle B_2\rangle_E$, $\langle B_3\rangle_E$, $\langle B_5\rangle_E$, $\langle B_4\rangle_E$	1
(B_2,B_3)	$\langle B_0,B_1\rangle_0$, $\langle B_2,B_3\rangle_1$, $\langle B_5\rangle_E$, $\langle B_4\rangle_E$	2
(B_4,B_3)	$\langle B_0,B_1\rangle_0$, $\langle B_2,B_3\rangle_1$, $\langle B_5\rangle_E$, $\langle B_4\rangle_E$	3
(B_1,B_2)	$\langle B_0,B_1,B_2,B_3\rangle_0$, $\langle B_5\rangle_E$, $\langle B_4\rangle_E$	4
(B_0,B_5)	$\langle B_0,B_1,B_2,B_3\rangle_0$, $\langle B_5\rangle_4$, $\langle B_4\rangle_E$	5
(B_5,B_4)	$\langle B_0,B_1,B_2,B_3\rangle_0$, $\langle B_5,B_4\rangle_4$	6
(B_1,B_4)	$\langle B_0,B_1,B_2,B_3\rangle_0$, $\langle B_5,B_4\rangle_4$	7

(b) Steps of the Hot-Path Building Algorithm on the Example

■ **FIGURE 8.18** Example of the Hot-Path Building Algorithm.

The compiler can use a greedy algorithm to find hot paths, as shown in Fig. 8.17. To begin, it creates a degenerate chain for each block that contains exactly that block. It sets the priority for each degenerate chain to a large number, such as the number of edges in the CFG or the largest integer.

Next, the algorithm iterates over the edges in the CFG and builds up chains to model the hot paths. It takes the edges in order of decreasing execution frequency. For an edge, (x, y), the algorithm merges the chain containing x with the chain containing y if and only if x is the last node in its chain and y is the first node in its chain. If either condition is not true, it leaves the chains that contain x and y alone.

The algorithm ignores self loops, (x, x), because they do not affect placement decisions.

If the algorithm merges the chains for x and y, it must assign the new chain an appropriate priority. It computes that priority as the minimum of the priorities of the chains for x and y. If both x and y are degenerate chains with their initial high priority, the algorithm sets the priority of the new chain to the ordinal number of merges that it has considered, denoted as P. The effect of this choice is to place the new chain behind chains constructed from higher-frequency edges and ahead of those constructed from lower-frequency edges.

The algorithm halts after it examines every edge. It produces a set of chains that model the hot paths in the CFG. Each node belongs to exactly one chain. Edges in chains execute more often than edges that cross from one chain to another. The priority values of the chains approximate an order that would maximize the number of executed forward branches.

Forward branch

A *forward branch* is one whose target has a higher address than its source. In some ISAs, forward branches are faster than backward branches.

To illustrate the algorithm's operation, consider its behavior when applied to the example CFG from the previous section, which appears in Fig. 8.18(a).

```
$t \leftarrow$ chain headed by the CFG entry node
WorkList $\leftarrow \{ (t, priority(t)) \}$
while (Worklist $\neq \emptyset$) do
    remove a chain c of lowest priority from WorkList
    for each block x in c in chain order do
        place x at the end of the executable code
    for each block x in c do
        for each edge $\langle x, y \rangle$ where y is unplaced do
            $t \leftarrow$ chain containing y
            if $(t, priority(t)) \notin$ WorkList then
                WorkList $\leftarrow$ WorkList $\cup \{ (t, priority(t)) \}$
```

■ **FIGURE 8.19** Code-Layout Algorithm.

Panel (b) shows the algorithm's progress on the example. Priorities are shown as subscripts on chains. E is $|edges|$.

Tie breaking among equal-priority chains can lead to different layouts. For example, if the algorithm chooses (B_4, B_3) before (B_2, B_3), then it builds the chains: $\langle B_0, B_1, B_2 \rangle_0$ and $\langle B_5, B_4, B_3 \rangle_1$. Different chains may produce different layouts. The algorithm still produces good results, even with a nonoptimal ordering for the equal-weight edges.

Performing Code Layout

The set of chains produced by the hot-path algorithm constitutes a partial order on the set of basic blocks. To produce the executable code, the compiler must place all of the blocks into a fixed linear order. Fig. 8.19 shows an algorithm that computes a linear layout from the set of chains. It encodes two simple heuristics: (1) place the blocks of a chain in order, so that fall-through branches implement the chain's edges, and (2) place chains in priority order, lowest to highest.

The implementation can use a sparse set for the worklist (see Appendix B.2.3).

The algorithm represents a chain as a pair, (l, p) where l is a list of blocks and p is the chain's priority. Each chain should enter the worklist exactly once. The following table shows the algorithm's behavior on the set of chains derived in Fig. 8.18(b):

Step	WorkList	Code Layout
—	$\langle B_0, B_1, B_2, B_3 \rangle_0$	
1	$\langle B_5, B_4 \rangle_4$	B_0, B_1, B_2, B_3
2	$\emptyset$	$B_0, B_1, B_2, B_3, B_5, B_4$

The first line shows the initial state; the worklist starts with the chain that contains B_0. The algorithm removes that chain from the worklist and places all its blocks, in order. Next, it processes the edges that leave the already placed blocks. These edges add the chain $\langle B_5, B_4 \rangle$ to the worklist.

The second iteration removes $\langle B_5, B_4 \rangle$ from the worklist and places B_5 and B_4 at the end of the layout. It finds no edges that lead to unplaced blocks, so the algorithm halts.

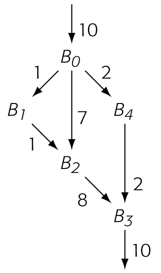
If, instead, the algorithm had produced the chains $\langle B_0, B_1, B_2 \rangle_0$ and $\langle B_5, B_4, B_3 \rangle_1$, the final layout would have been different.

Step	WorkList	Code Layout
—	$\langle B_0, B_1, B_2 \rangle_0$	
1	$\langle B_5, B_4, B_3 \rangle_1$	B_0, B_1, B_2
2		$B_0, B_1, B_2, B_5, B_4, B_3$

If we assume that the estimated execution frequencies are correct, there is no reason to prefer one layout over the other.

A Final Example

As a final example, consider the CFG shown in the margin. The first step in the placement algorithm is to apply the hot-path algorithm. It proceeds as follows:



Final Example

Edge	Set of Chains	P
—	$\langle B_0 \rangle_E$, $\langle B_1 \rangle_E$, $\langle B_2 \rangle_E$, $\langle B_3 \rangle_E$, $\langle B_4 \rangle_E$	0
(B_2, B_3)	$\langle B_0 \rangle_E$, $\langle B_1 \rangle_E$, $\langle B_2, B_3 \rangle_0$, $\langle B_4 \rangle_E$	1
(B_0, B_2)	$\langle B_0, B_2, B_3 \rangle_0$, $\langle B_1 \rangle_E$, $\langle B_4 \rangle_E$	2
(B_0, B_4)	$\langle B_0, B_2, B_3 \rangle_0$, $\langle B_1 \rangle_E$, $\langle B_4 \rangle_2$	3
(B_4, B_3)	$\langle B_0, B_2, B_3 \rangle_0$, $\langle B_1 \rangle_E$, $\langle B_4 \rangle_2$	4
(B_0, B_1)	$\langle B_0, B_2, B_3 \rangle_0$, $\langle B_1 \rangle_4$, $\langle B_4 \rangle_2$	5
(B_1, B_2)	$\langle B_0, B_2, B_3 \rangle_0$, $\langle B_1 \rangle_4$, $\langle B_4 \rangle_2$	6

The algorithm halts with one multinode chain and two degenerate chains. It captures the fact that $\langle B_0, B_2\ B_3 \rangle$ accounts for most of the executions and that both B_4 and B_1 execute infrequently.

The layout algorithm first places $\langle B_0, B_2, B_3 \rangle$. When it processes the outbound edges from those nodes, it adds both of the degenerate blocks to the worklist. The next two iterations remove and place B_4 and then B_1.

SECTION REVIEW

Optimizations that examine an entire procedure have opportunities for improvement that are not available at smaller scopes. Because the global, or whole-procedure, scope includes cyclic paths, global optimizations usually need global analysis. As a consequence, these algorithms typically have an offline flavor; they analyze the code before they transform it.

This section highlighted two distinct kinds of analysis: global data-flow analysis and runtime collection of profile data. Data-flow analysis is a compile-time technique that accounts, mathematically, for the effects along all possible paths through the code. By contrast, profile data records what actually happened on a single run of the code, with a single set of input data. Data-flow analysis is conservative, in that it accounts for all possibilities. Runtime profiling is aggressive, in that it assumes that future runs will share runtime characteristics with the profiling run. Both kinds of information can play an important role in optimization.

REVIEW QUESTIONS

1. In some situations, the compiler needs to know that a variable is live along *all* paths that leave a block, rather than live along *some* path. Reformulate the equations for LIVEOUT so that they compute the set of names that are used before definition along every path from the end of the block to the CFG's exit node, n_f.
2. To collect accurate edge-count profiles, the compiler can instrument each edge in the profiled procedure's CFG. A clever implementation can instrument a subset of those edges and deduce the counts for the rest. Devise a scheme that derives accurate edge-count data without instrumenting each branch. On what principles does your scheme rely?

8.7 INTERPROCEDURAL OPTIMIZATION

As discussed in Chapter 6, procedure calls form boundaries in software systems. The division of a program into multiple procedures has both positive and negative impacts on the compiler's ability to generate efficient code. On the positive side, it limits the amount of code that the compiler considers at any one time. This effect keeps compile-time data structures small and limits the cost of compile-time algorithms by limiting the problem sizes.

On the negative side, procedure calls make compile-time knowledge less precise. For example, consider a call from `fee` to `fie` that passes a variable x as a call-by-reference parameter. If the compiler knows that x has the value 15 before the call, it cannot use that fact after the call, unless it knows that the call cannot change *x*. To use *x*'s value after the call, the compiler must prove that the formal parameter corresponding to *x* is not modified by `fie` or any procedure `fie` calls, directly or indirectly.

Procedure calls also introduce direct costs in the precall, postreturn, prolog, and epilog sequences (see Section 6.5). These sequences consist of multiple operations, each of which takes time to execute. The transitions between these sequences require jumps. In the general case, all of these operations are needed. For any specific call, however, the compiler may be able to tailor either the sequences or the body of the callee to the actual context and achieve better performance.

Procedures and procedure calls introduce inefficiencies that intraprocedural optimization cannot address. To reduce some of these inefficiencies, the compiler can analyze and transform multiple procedures together, using interprocedural analysis and optimization. These techniques are equally important in Algol-like languages and in object-oriented languages.

Interprocedural data-flow analysis is discussed in Sections 9.2.4 and 9.4.

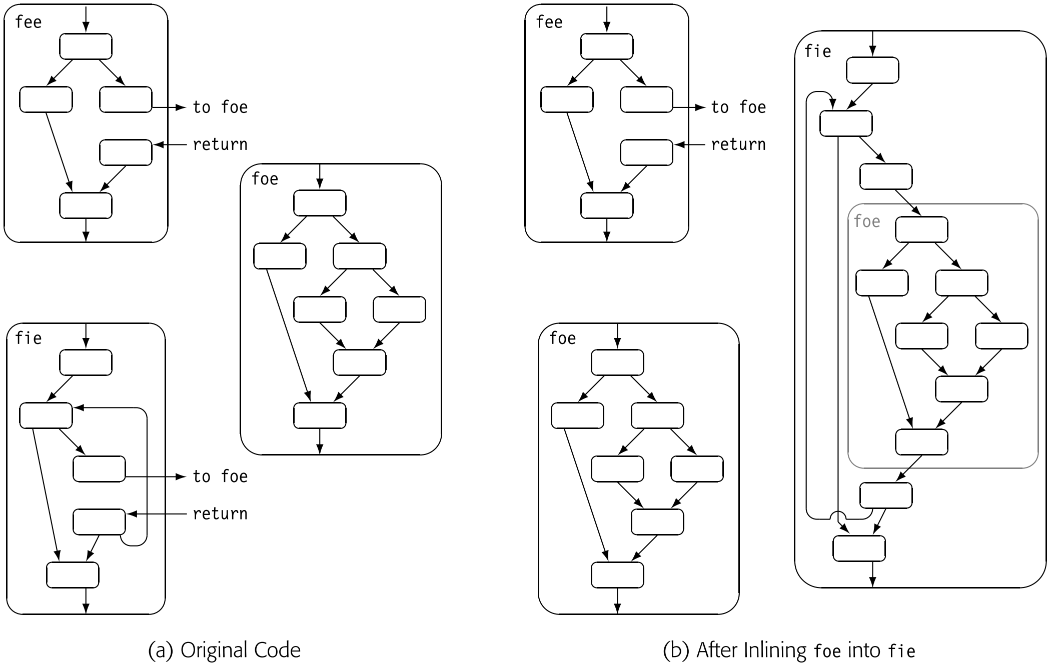
This section presents two different interprocedural optimizations: inline substitution of procedure calls and procedure placement for improved code locality. Because whole-program optimization requires that the compiler has access to the code being analyzed and transformed, the decision to perform whole-program optimization has implications for the structure of the compiler. Thus, the final subsection discusses the structural issues that arise in a system that includes interprocedural analysis and optimization.

8.7.1 Inline Substitution

As we saw in Chapters 6 and 7, procedure linkage code involves a significant number of operations. The linkage code allocates an activation record, evaluates each actual parameter, preserves the caller's state, creates the callee's environment, transfers control from the caller to the callee and back, restores the caller's state, and, if necessary, returns values from the callee to the caller. These runtime actions are part of the overhead of using a high-level language; they maintain programming-language abstractions but are not strictly needed to compute the results. Optimizing compilers try to reduce the cost of such overheads.

In some cases, the compiler can improve the efficiency of the final code by replacing a call site with a copy of the callee's body, appropriately tailored to the context at the call site. This transformation, called *inline substitution*, lets the compiler eliminate much of the code in the linkage sequences and tailor the new copy of the callee's body to the caller's context. Because

Inline substitution
a transformation that replaces a call site with a copy of the callee's body, rewritten to reflect the parameter bindings



■ **FIGURE 8.20** Inline Substitution Example.

inline substitution moves code from one procedure to another and alters the program's call graph, it is inherently an interprocedural transformation.

The implementation of inline substitution partitions naturally into two subproblems: the actual transformation and a decision procedure that chooses which call sites to inline. The transformation itself is relatively simple. The decision procedure is more complex. The specific choices have a direct impact on the transformation's effectiveness.

The Transformation

To perform inline substitution, the compiler rewrites a call site with the body of the callee, while making appropriate modifications to model the effects of parameter binding. Fig. 8.20(a) shows two procedures, `fee` and `fie`, both of which call a third procedure, `foe`. Panel (b) depicts the control flow after inlining the call from `fie` to `foe`. The compiler has created a copy of `foe` and moved it inside `fie`, connected `fie`'s precall sequence directly to the prolog of its internal copy of `foe` and connected the epilog to the postreturn sequence in a similar fashion. Some of the resulting blocks can be merged, enabling improvement with subsequent optimization.

Of course, the compiler must use an IR that can represent the inlined procedure. Some source-language constructs can create arbitrary and unusual control-flow constructs in the resulting code. For example, a callee with multiple premature returns may generate a control-flow graph (CFG) that is hard to represent in a near-source IR.

In the transformation, the compiler writer should pay attention to the proliferation of local variables. A simple implementation would create one new local variable in the caller for each local variable in the callee. If the compiler inlines several procedures, or several call sites to the same callee, the local name space can grow quite large. While growth in the name space is not a correctness issue, it can significantly increase the compile time of transformed code.

If the compiler uses data-flow analysis, growth in the name space appears as growth in the sizes of the sets used to solve data-flow problems. A typical data-flow problem uses several sets per node in the CFG.

The Decision Procedure

Choosing which call sites to inline is a complex task. Inlining a given call site can improve performance; unfortunately, it can also degrade performance. To make intelligent choices, the compiler must consider a broad range of characteristics of the caller, the callee, and the call site. The compiler must also understand its own strengths and weaknesses.

The primary sources of improvement from inlining are direct elimination of operations and improved effectiveness of other optimizations. The former effect can occur when parts of the linkage sequence can be eliminated; for example, register save and restore code might be eliminated in favor of allowing the register allocator make those decisions. Knowledge from the caller may prove code in the callee to be dead or useless. More precise knowledge may also expose constant values or redundancies.

The primary source of degradation from inline substitution is decreased effectiveness of code optimization on the resulting code. Inlining the callee can increase code size and name space size. It can increase demand for registers in the neighborhood of the original call site. Eliminating the register save and restore code changes the problem seen by the register allocator. In practice, any of these can lead to a decrease in optimization effectiveness.

Changes in architecture, such as larger register sets, can increase the cost of a procedure call. That change can, in turn, make inlining more attractive.

At each call site, the compiler must decide whether or not to inline the call. To complicate matters, a decision made at one call site affects the decision at other call sites. For example, if *a* calls *b* which calls *c*, choosing to inline *c* into *b* changes both the characteristics of the procedure that might be inlined into *a* and the call graph of the underlying program. Furthermore, inlining has effects, such as code size growth, that must be viewed across the whole program; the compiler writer may want to limit the overall growth in code size.

Inline any call site that matches one of the following:

(1) The callee uses more than t_0 percent of execution time, and
- *(a) the callee contains no calls, or*
- *(b) the static call count is one, or*
- *(c) the call site has more than t_1 constant-valued parameters*

(2) The call site represents more than t_2 percent of all calls, and
- *(a) the callee is smaller than t_3, or*
- *(b) inlining the call will produce a procedure smaller than t_4*

■ **FIGURE 8.21** A Typical Decision Heuristic for Inline Substitution.

Decision procedures for inline substitution examine a variety of criteria at each call site. These include:

- *Callee Size* If the callee is smaller than the code in the linkage sequence, then inlining the callee should reduce both the code size and the number of operations executed. This situation arises surprisingly often.
- *Caller Size* The compiler may limit the overall size of any procedure to mitigate increases in compile time and decreases in optimization effectiveness.
- *Dynamic Call Count* An improvement to a frequently executed call provides greater benefit than the same improvement to a rarely executed call. In practice, compilers use either profile data or simple estimates, such as 10 times the call site's loop nesting depth.
- *Constant-Valued Actual Parameters* If some actual parameters have known constant values, the transformed code can fold those constants directly into the callee's body.
- *Static Call Count* Compilers often track the number of distinct sites that call a procedure. Any procedure called from just one call site can be inlined without any code space growth.
- *Parameter Count* The number of parameters can serve as a proxy for the cost of the procedure linkage, as the compiler must generate code to evaluate and store each actual parameter.
- *Calls in the Procedure* Tracking the number of calls in a procedure provides an easy way to detect leaves in the call graph. Leaf procedures are often good candidates for inlining.
- *Loop Nesting Depth* Call sites in loops execute more frequently than call sites outside loops. They also disrupt the compiler's ability to schedule the loop as a single unit (see Section 12.4).
- *Fraction of Execution Time* Computing the fraction of execution time spent in each procedure from profile data can prevent the compiler from inlining routines that cannot have a significant impact on performance.

As inlining proceeds, the compiler may need to update these metrics. For example, inlining b into a will change both the *static call count* metric for b and the *calls in the procedure* metric for a.

In practice, compilers precompute some or all of these metrics and then apply a set of heuristics to determine which call sites to inline. Fig. 8.21 shows a typical heuristic. It relies on a series of threshold parameters, named t_0 through t_4. The specific values chosen for the parameters will govern much of the heuristic's behavior; for example, t_3 should have a value greater than the size of the standard precall and postreturn sequences. The best settings for some of these parameters are almost certainly program specific.

8.7.2 Procedure Placement

The global code placement technique in Section 8.6.2 rearranged blocks within a single procedure. The analogous whole-program problem is to rearrange the procedures in an executable:

> *Given the call graph for a program, annotated with execution frequencies for each call site, rearrange the procedures to reduce virtual-memory working-set sizes and to limit the potential for call-induced conflicts in the instruction cache.*

The principle is simple. If procedure p calls procedure q and the compiler places them next to each other in memory, then they should not conflict in the instruction cache unless $size(p) + size(q)$ is greater than the cache size. Adjacent placement also increases the likelihood that p and q are located in the same virtual memory page, which can reduce working set size and page faults. Thus, the algorithm tries to maximize the number of calls for which the caller and callee are adjacent.

Virtual-to-physical address mapping may complicate that statement, but cache associativity should mitigate the issue.

To compute a placement, the algorithm treats the program's call graph as a set of constraints on the relative placement of procedures in the executable code. Each call-graph edge, (p,q), specifies an adjacency that should occur in the executable code. Unfortunately, the compiler cannot satisfy all of those adjacencies. For example, if p calls q, r, and s, the compiler cannot place all three of them next to p. The algorithm uses a greedy approximate technique to find a good placement, rather than trying to compute an optimal placement.

Recall that a program's call graph has a node for each procedure and an edge (x,y) for each call from x to y.

Procedure placement differs subtly from the global code placement problem discussed in Section 8.6.2. The global algorithm improves the code by ensuring that hot paths can be implemented with fall-through branches. Thus, the chain-construction algorithm in Fig. 8.17 ignores any CFG edge unless

```
// Initialization work
build the call multi-graph G
initialize Q as a priority queue          // Order Q highest to lowest

for each edge (x,y) ∈ G do                // Add weights to the edges
    if (x = y) then                       // Self loop is irrelevant
        delete (x,y) from G
    else
        weight( (x,y) ) ← estimated execution frequency for (x,y)

for each node x ∈ G do
    list(x) ← {x}                         // Initialize placement lists

    if multiple edges exist from x to y then
        combine them and their weights

    for each edge (x,z) ∈ G do            // Put each edge into Q
        Enqueue(Q, (x,z), weight( (x,z) ))

// Iterative reduction of the graph
while Q is not empty do
    (x,y) ← Dequeue(Q)                    // Take highest priority edge

    for each edge (y,z) ∈ G do            // Move source from y to x
        ReSource( (y,z), x)

    for each edge (z,y) ∈ G do            // Move target from y to x
        ReTarget( (z,y), x)

    append list(y) to list(x)             // Update the placement list

    delete y and its edges from G         // Clean up G
```

■ **FIGURE 8.22** Analysis Phase of the Procedure Placement Algorithm.

it runs from the tail of one chain to the head of another. By contrast, as the procedure placement algorithm builds chains of procedures, it can use edges that run between procedures that lie in the middles of their chains because its goal is simply to place procedures near each other—to reduce working set sizes and to reduce interference in the instruction cache. If p calls q and the distance from p to q is less than the cache size, placement achieves its objectives. Thus, in some sense, the procedure placement algorithm has more freedom than the block-layout algorithm.

Procedure placement consists of two phases: analysis and transformation. The analysis operates on the program's call graph. It repeatedly selects two call graph nodes that are connected by an edge and combines them. The order of combination is driven by execution frequency data, either measured or estimated. The order of combination determines the final layout. The layout phase is straightforward; it simply rearranges the code for the procedures into the order chosen by the analysis phase.

Fig. 8.22 shows a greedy algorithm for the analysis phase of procedure placement. It constructs a placement by considering call-graph edges in order of decreasing execution frequency. As a first step, it builds the call graph, annotates each edge with its estimated execution frequency, and combines all the edges between two nodes into a single edge. As the final part of its initialization work, it builds a priority queue of the call-graph edges, ordered by their weights.

When the algorithm combines edges, the new edge receives the sum of the old edges' weights.

The second half of the algorithm iteratively constructs an order for procedure placement. The algorithm associates with each node in the graph an ordered list of procedures. These lists specify a linear order among the named procedures. When the algorithm halts, the lists will specify a total order on the procedures in each node that can be used to place them in the executable code.

The algorithm uses the call-graph edge weights to guide the process. It repeatedly selects the highest-weight edge, say (x,y), from the priority queue and combines its source x and its sink y. Next, it must update the call graph to reflect the change.

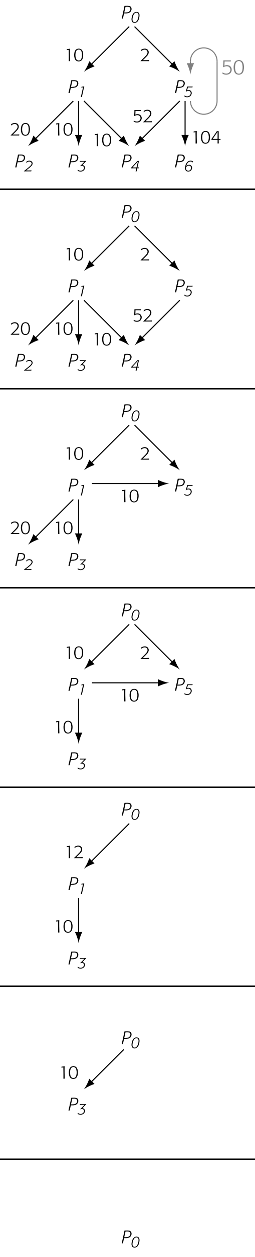
1. For each edge (y,z), it calls *ReSource* to replace (y,z) with (x,z) and to update the priority queue. If (x,z) already exists, *ReSource* combines the execution counts of (y,z) and the existing (x,z).
2. For each edge (z,y), it calls *ReTarget* to replace (z,y) with (z,x) and to update the priority queue. If (z,x) already exists, *ReTarget* combines the execution counts of (z,y) and the existing (z,x).

To force the placement of y after x, the algorithm appends *list*(y) to *list*(x). Finally, it deletes y and its edges from the call graph.

The algorithm halts when the priority queue is empty. The final graph has one node for each of the connected components of the original graph. If all nodes are reachable from the program's entry, the final graph consists of a single node. If some nodes are not reachable, either because no path in the program calls them or because those paths are obscured by ambiguous calls, then the final graph will consist of multiple nodes. Either way, the compiler and linker can use the lists associated with nodes in the final graph to specify the relative placement of procedures.

Example

To see how the procedure placement algorithm works, consider the example call graph shown in panel 0 of Fig. 8.23. The edge from P_5 to itself is shown in gray because it only affects the algorithm by changing the execution



Initial State

0.

	P_0	P_1	P_2	P_3	P_4	P_5	P_6
List(x):	$\{P_0\}$	$\{P_1\}$	$\{P_2\}$	$\{P_3\}$	$\{P_4\}$	$\{P_5\}$	$\{P_6\}$

Q: $\{(P_5,P_6)_{104}, (P_5,P_4)_{52}, (P_1,P_2)_{20}, (P_1,P_4)_{10}, (P_1,P_3)_{10}, (P_0,P_1)_{10}, (P_0,P_5)_2\}$

1.

	P_0	P_1	P_2	P_3	P_4	P_5
List(x):	$\{P_0\}$	$\{P_1\}$	$\{P_2\}$	$\{P_3\}$	$\{P_4\}$	$\{P_5,P_6\}$

Q: $\{(P_5,P_4)_{52}, (P_1,P_2)_{20}, (P_1,P_4)_{10}, (P_1,P_3)_{10}, (P_0,P_1)_{10}, (P_0,P_5)_2\}$

2.

	P_0	P_1	P_2	P_3	P_5
List(x):	$\{P_0\}$	$\{P_1\}$	$\{P_2\}$	$\{P_3\}$	$\{P_5,P_6,P_4\}$

Q: $\{(P_1,P_2)_{20}, (P_1,P_5)_{10}, (P_1,P_3)_{10}, (P_0,P_1)_{10}, (P_0,P_5)_2\}$

3.

	P_0	P_1	P_3	P_5
List(x):	$\{P_0\}$	$\{P_1,P_2\}$	$\{P_3\}$	$\{P_5,P_6,P_4\}$

Q: $\{(P_1,P_5)_{10}, (P_1,P_3)_{10}, (P_0,P_1)_{10}, (P_0,P_5)_2\}$

4.

	P_0	P_1	P_3
List(x):	$\{P_0\}$	$\{P_1,P_2,P_5,P_6,P_4\}$	$\{P_3\}$

Q: $\{(P_0,P_1)_{12}, (P_1,P_3)_{10}\}$

5.

	P_0	P_3
List(x):	$\{P_0,P_1,P_2,P_5,P_6,P_4\}$	$\{P_3\}$

Q: $\{(P_0,P_3)_{10}\}$

6.

	P_0
List(x):	$\{P_0,P_1,P_2,P_5,P_6,P_4,P_3\}$

Q: $\emptyset$

■ **FIGURE 8.23** Steps of the Procedure Placement Algorithm.

frequencies. A self-loop cannot affect placement since its source and sink are identical.

Panel 0 shows the state of the algorithm immediately before the iterative reduction begins. Each node has the trivial list that contains its own name. The priority queue has every edge, except the self loop, ranked by execution frequency.

Panel 1 shows the state of the algorithm after the first iteration of the while loop. The algorithm collapsed P_6 into P_5, and updated both the list for P_5 and the priority queue.

In panel 2, the algorithm has collapsed P_4 into P_5. It retargeted (P_1,P_4) onto P_5 and changed the corresponding edge name in the queue. It also removed P_4 from the graph and updated the list for P_5.

The other iterations proceed in a similar fashion. Panel 4 shows a situation where the algorithm combined edges. When it collapsed P_5 into P_1, it retargeted (P_0,P_5) onto P_1. Since (P_0,P_1) already existed, it simply combined their weights and updated the priority queue by deleting (P_0,P_5) and changing the weight on (P_0,P_1).

At the end of the iterations, the graph has been collapsed to a single node, P_0. While this example constructed a layout that begins with the entry node, that happened because of the edge weights rather than by algorithmic design.

8.7.3 **Pragmatics of Interprocedural Optimization**

Building a compiler that performs analysis and optimization across two or more procedures fundamentally changes the relationship between the compiler and the code that it produces. Traditional compilers have compilation units of a single procedure, a single class, or a single file of code; the resulting code depends solely on the contents of that compilation unit. Once the compiler uses knowledge about one procedure to optimize another, the correctness of the resulting code depends on the state of both procedures.

Compilation unit
The portion of a program presented to the compiler is often called a *compilation unit*.

Consider the impact of inline substitution on the validity of the optimized code. Assume that the compiler inlines `fie` into `fee`. Any subsequent editing change to `fie` will necessitate recompilation of `fee`—a dependence that results from an optimization decision rather than from any relationship exposed in the source code.

If the compiler collects and uses interprocedural information, similar problems can arise. For example, `fee` may call `fie`, which calls `foe`; assume that the compiler relies on the fact that the call to `fie` does not change the known constant value of the global variable x. If the programmer subsequently edits foe so that it modifies x, that change can invalidate the prior compilations of both fee and fie by changing the facts upon which optimization relies. Thus, a change to foe can necessitate a recompilation of other procedures in the program.

To make interprocedural optimization practical, manage recompilation, and provide the necessary source code access, researchers have proposed a variety of different scenarios. These include using larger compilation units, integrating the compiler into a development environment, performing link-time optimization, and relying on just-in-time compilation for interprocedural optimizations.

Enlarging Compilation Units

The simplest solution to the practical problems introduced by interprocedural optimization is to enlarge the compilation units. If the compiler only considers optimization and analysis within a compilation unit, it can sidestep the issue of recompilation. It can limit its analysis and optimization to code that is compiled together; then, it will not introduce dependences between compilation units and it should not require access to information about other units. The IBM PL/I optimizing compiler took this approach; code quality improved as related procedures were aggregated into the same file.

Of course, this approach limits the opportunities for interprocedural optimization. It also encourages the programmer to create larger compilation units and to group together procedures that call one another. Both of these may introduce practical problems in a system with multiple programmers. Still, as a practical matter, this organization is attractive because it is a minor change from traditional models of compilation.

Integrated Development Environments

If the compiler is embedded inside an integrated development environment (IDE), the compiler can access code as needed through the IDE. The IDE can notify the compiler when source code changes so that the compiler can determine if recompilation is needed. This model shifts ownership of both the source code and the compiled code from the developer to the IDE. Collaboration between the IDE and the compiler then ensures that appropriate actions are taken to guarantee consistent and correct optimization.

Link-Time Optimization

The compiler writer can shift interprocedural optimization into the linker, where it will have access to all of the *statically* linked code. To obtain the full benefits of this approach, the linker may also need to perform subsequent global optimization. Since the results of link-time optimization are only recorded in the executable, and that executable is discarded on the next compilation, this strategy sidesteps the recompilation problem. It almost certainly performs more analysis and optimization than the other approaches, but it offers both simplicity and obvious correctness.

Just-In-Time Compilation

An environment that performs runtime optimization faces a different set of opportunities and constraints (see Chapter 14). At runtime, all of the code for the application is available; there is no issue with access. The just-in-time compiler can optimize across procedure boundaries. The system discards JIT-compiled code at the end of execution, so recompilation concerns only arise from changes in the code made during execution—changes that can be handled through careful deoptimization (see the discussion on page 742).

A hot-trace optimizer can easily build traces that cross procedure calls (see Section 14.3). Hot-method optimizers often include inline substitution to lower the cost of dynamic dispatch (see Section 14.4).

At the same time, JITs operate under time consraints that almost certainly prevent the JIT from applying classic whole-program analyses such as computing summary information or performing interprocedural constant propagation. Runtime optimization systems tend to be profile driven; they are more likely to find and optimize specific hot calls than they are to discover that some subpart of the call tree is side-effect free. Despite these limitations, JIT-based systems can profit from selective application of interprocedural optimizations. These systems do use interprocedural methods; they do so precisely because it is profitable.

SECTION REVIEW

Analysis and optimization across procedure boundaries can reveal new opportunities for code improvement. Examples include tailoring the procedure linkage (precall, prolog, epilog, and postreturn sequences) to call-site specific constant values, or laying out procedures in memory to improve locality. Many techniques have been proposed to recognize and exploit interprocedural opportunities; inline substitution is one of the best known and most broadly effective of these techniques.

A compiler that applies interprocedural techniques must take care to ensure that the executables it builds are based on a consistent view of the entire program. Using facts from one procedure to modify the code in another can introduce subtle dependences between the code in distant procedures, dependences that the compiler must recognize and respect. Multiple strategies have been proposed to mitigate these effects and demonstrated in practical systems.

REVIEW QUESTIONS

1. Suppose procedure a invokes b and c. If the compiler inlines the call to b, what code space and data space savings might arise? If it inlines c as well, are further data-space savings possible?
2. In procedure placement, what happens to a procedure whose incoming edges all have estimated execution frequencies of zero? Where should the algorithm place such a procedure? Does the treatment of such a procedure affect execution time performance? Can the compiler eliminate them as useless?

8.8 SUMMARY AND PERSPECTIVE

The optimizer in a typical compiler contains a collection of techniques that try to improve the performance of the compiled code. While most optimizations try to improve runtime speed, optimizations can also target other measures, such as code size or energy consumption. This chapter has shown a variety of techniques that operate over scopes that range from single basic blocks through entire programs.

Optimizations improve performance by tailoring general translation schemes to the specific details of the code at hand. The transformations in an optimizer try to remove the overhead introduced in support of source-language abstractions, including data structures, control structures, and error checking. They try to recognize special cases that have efficient implementations and rewrite the code to realize those savings. They try to match the resource needs of the program against the actual resources available on the target processor, including functional units, the capacity and bandwidth of each level in the memory hierarchy (registers, cache, translation lookaside buffers, and memory), and instruction-level parallelism.

Before the optimizer can apply a transformation, it must determine that the proposed rewrite of the code is safe—that it preserves the code's original meaning. Typically, the optimizer must analyze the code to prove safety. In this chapter, we saw a number of approaches to proving safety, ranging from the bottom-up construction of the value table in local value numbering through computing LiveOut sets to detect uninitialized variables.

Once the optimizer has determined that it can safely apply a transformation, it must decide whether or not the rewrite will improve the code. Some techniques, such as local value numbering, simply assume that the rewrites they use are profitable. Other techniques, such as inline substitution, require complicated decision procedures to determine when a transformation might improve the code.

This chapter provided a basic introduction to the field of compiler-based code optimization. It introduced many of the terms and issues that arise in optimization. It does not include an "Advanced Topics" section; instead, the interested reader will find additional material on static analysis in support of optimization in Chapter 9 and on optimizing transformations in Chapter 10.

CHAPTER NOTES

The field of code optimization has a long and detailed literature. For a deeper treatment, the reader should consider some of the specialized books on the subject [21,277,279]. It would be intellectually pleasing if code optimization had developed in a logical and disciplined way, beginning with local techniques, extending them first to regions, then to entire procedures, and finally to entire programs. As it happened, however, development occurred in a more haphazard fashion. For example, the original Fortran compiler [28] performed both local and global optimization—the former on expression trees and the latter for register allocation. Interest in both regional techniques, such as loop optimization [260], and interprocedural techniques, such as inline substitution, crops up early in the literature, as well [17].

Local value numbering, with its extensions for algebraic simplifications and constant folding, is usually credited to Balke in the late 1960s [17,94], although it is clear that Ershov achieved similar effects in a much earlier system [150]. Similarly, Floyd mentioned the potential for both local redundancy elimination and application of commutativity [160]. The extension to EBBs in superlocal value numbering is natural and has, undoubtedly, been invented and reinvented in many compilers. Our treatment derives from Simpson [59].

The tree-height balancing algorithm is due to Hunt and McKinley [105]; it uses a rank function inspired by Huffman codes, but is easily adapted to other metrics. The classic algorithm for balancing instruction trees is due to Baer and Bovet [30]. In practice, the tree-height balancing algorithm might be best applied just before scheduling.

Loop unrolling is the simplest loop nest optimization. It has a long history in the literature [17]. Several authors have studied selection of unroll factors [130,337]. The use of unrolling to eliminate register-to-register copy operations as in Review Question 2 for Section 8.5 is from Kennedy [225]. Unrolling can have subtle and surprising effects [118].

The ideas that underlie live analysis have been around as long as compilers have been automatically allocating storage locations for values [250]. Beatty first defined live analysis in an internal IBM technical report [16]. Lowry and Medlock discuss "busy" variables [260] and the use of this information in both dead-code elimination and reasoning about interference (see Chapter 13). The formulation of liveness as a global data-flow problem appeared by 1971 [14,224]. It plays a key role in many optimizations (see, for example, Sections 9.3 and 13.4).

The code-placement algorithms, at both the global and whole-program scopes, are taken from Pettis and Hansen [293]. Subsequent work on this problem has focused on collecting better profile data and improving the placements [171,194]. Later work includes work on branch alignment [72,369] and code layout [85,100,171].

Interprocedural optimization has been discussed in the literature for decades [17,334]. While inline substitution, as a transformation, is straightforward, its profitability has been the subject of many studies [34,108,129,309]. Hall reports on one study where increased name space size led directly to a decrease in effectiveness of optimization [108]. All of the scenarios mentioned in Section 8.7.3 have been explored in real systems [114,334,354]. Interprocedural analysis has a long and rich history [19,35,37,334]. Recompilation analysis is treated in depth by Burke and Torczon [70,347]. See the notes for Chapter 9 for more references on interprocedural analysis. The interprocedural aspects of JIT compilation are discussed in more depth in Chapter 14.

EXERCISES

Section 8.4

1. Apply the algorithm from Fig. 8.4 to blocks B_0 and B_1.

```
t1 ← a + b
t2 ← t1 + c
t3 ← t2 + d
t4 ← b + a
t5 ← t3 + e
t6 ← t4 + f
t7 ← a + b
t8 ← t4 - t7
t9 ← t8 * t6
```

Block B_0

```
t1 ← a × b
t2 ← t1 × 2
t3 ← t2 × c
t4 ← 7 + t3
t5 ← t4 + d
t6 ← t5 + 3
t7 ← t4 + e
t8 ← t6 + f
t9 ← t1 + 6
```

Block B_1

Exposed value is defined on page 404.

2. Design an algorithm to convert a block of three-address code into a forest of expression trees, with the specification that each *exposed* value is the root of its own tree. You may assume that you have LiveOut and LiveIn sets for the block.

3. Consider a basic block b, such as B_0 or B_1 in Exercise 8.1. It has n operations, numbered from 1 to n.

   a. For a name x, USES(x) contains the index in b of each operation that uses x as an operand. Write an algorithm to compute the USES set for every name mentioned in block b.

   b. Apply your algorithm to blocks B_0 and B_1 from Exercise 8.1.

   c. For a reference to x in operation i of block b, DEF(x,i) is the index in b of the operation that defines the value of x visible at operation i. Write an algorithm to compute DEF(x,i) for each reference x in b. If x is upward exposed at i, then DEF(x,i) should be -1.

   d. Apply your algorithm to blocks B_0 and B_1 from Exercise 8.1.

4. Apply the tree-height balancing algorithm from Section 8.4.2 to blocks B_0 and B_1 from Exercise 8.1.

   Assume LIVEOUT(B_0) = {t_3, t_9} and LIVEOUT(B_1) = {t_7, t_8, t_9}. The names a through f are upward-exposed in the blocks.

Section 8.5

5. For the control-flow graph shown in Fig. 8.24:

   a. Find the extended basic blocks and list their distinct paths.

   b. Apply local value numbering (LVN) to each block.

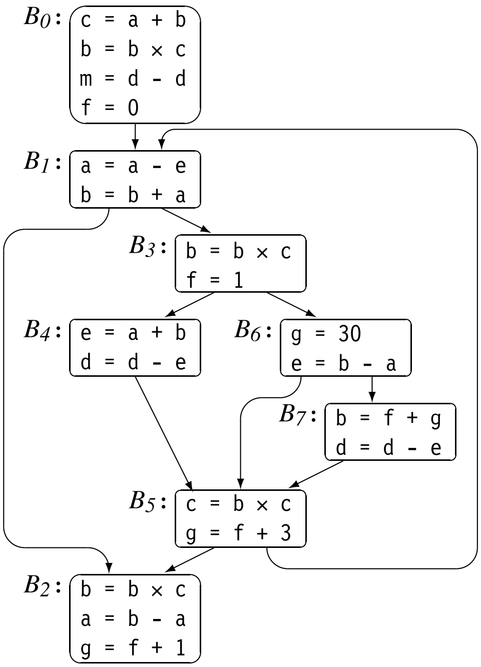
   c. Apply superlocal value numbering (SVN) to the EBBs. Note any improvements that it finds beyond those found by LVN.

6. Consider the following simple five-point stencil computation:

```
      do 20 i = 2, n-1, 1
         t1 = A(i,1)
         t2 = A(i,2)
         do 10 j = 2, m-1, 1
            t3 = A(i,j+1)
            A(i,j) = 0.2 × (t1 + t2 + t3
                     + A(i-1,j) + A(i+1,j))
            t1 = t2
            t2 = t3
   10    continue
   20 continue
```

   Each iteration of the loop executes two copy operations.

   a. Loop unrolling can eliminate the copy operations. What unroll factor is needed to eliminate all copy operations in this loop?



■ **FIGURE 8.24** Control-Flow Graph for Question 8.5.

b. In general, if a loop contains multiple cycles of copy operations, how can you compute the unroll factor needed to eliminate all of the copy operations?

Section 8.6

7. At some point *p*, LIVE(*p*) is the set of names that are *live* at *p*. LIVEOUT(*b*) is just the LIVE set at the end of block *b*.

   a. Develop an algorithm that takes as input a block *b* and its LIVEOUT set and produces as output the LIVE set for each operation in the block.

   b. Apply your algorithm to blocks B_0 and B_1 in Exercise 8.1. Assume that LIVEOUT(B_0) $= \{\texttt{t}_3, \texttt{t}_9\}$ and LIVEOUT(B_1) $= \{\texttt{t}_7, \texttt{t}_8, \texttt{t}_9\}$.

8. Compute LIVEOUT sets for each of the blocks in the control-flow graph shown in Fig. 8.24.

9. Fig. 8.17 shows an algorithm for finding the hot paths in a CFG.

   a. Devise an alternative hot-path construction that pays attention to ties among equal-weight edges.

   b. Construct two examples where your algorithm leads to a code layout that improves on the layout produced by the book's algorithm.

Use the code-layout algorithm from Fig. 8.19 with the chains constructed by your algorithm and those built by the book's algorithm.

Section 8.7

10. Consider the following psuedocode fragment. It shows a procedure `fee` and two call sites that invoke `fee`.

```
static int A[1000,1000], B[1000], sum;
    ...
x = A[i,j] + y;
call fee(i,j,1000);
    ...
call fee(1,1,0);
    ...
fee(int row; int col; int ub)
    int i;
    sum = A[row,col];
    for (i=0; i<ub; i++)
        sum = sum + B[i];
```

   a. What benefits would you expect from inlining `fee` at each of the call sites? Estimate the fraction of `fee`'s code that would remain after inlining and subsequent optimization.

   b. Based on your experience in part a, sketch a high-level algorithm to estimate the benefits of inlining a specific call site. Your method should consider the impact on both the caller and the callee.

11. In the previous problem, features of the call site and its context determined the extent to which the optimizer could improve the inlined code. Sketch, at a high level, a procedure for estimating the improvements that might accrue from inlining a specific call site. (With such an estimator, the compiler could inline the call sites with the highest estimated profit, stopping when it reached some threshold on procedure size or total program size.)

12. When the procedure placement algorithm, shown in Fig. 8.22, considers an edge (p,q) it always places p before q.

   a. Formulate a modification of the algorithm that would consider placing the sink of an edge before its source.

   b. Construct an example where this approach places two procedures closer together than the original algorithm. Assume that all procedures are of uniform size.

13. In the example from Fig. 8.23, there are three equal weight edges: (P_0,P_1), (P_1,P_3), and (P_1,P_4). Show how the algorithm would proceed if it took those edges in another order.

This page intentionally left blank

Chapter 9

Data-Flow Analysis

ABSTRACT

Compilers analyze the IR form of the program in order to identify opportunities where the code can be improved and to prove the safety and profitability of transformations that might improve that code. Data-flow analysis is the classic technique for compile-time program analysis. It allows the compiler to reason about the runtime flow of values in the program.

This chapter explores iterative data-flow analysis, based on a simple fixed-point algorithm. From basic data-flow analysis, it builds up to construction of static single-assignment (SSA) form, illustrates the use of SSA form, and introduces interprocedural analysis.

KEYWORDS

Data-Flow Analysis, Dominance, Static Single-Assignment Form, Constant Propagation

9.1 INTRODUCTION

As we saw in Chapter 8, optimization is the process of analyzing a program and transforming it in ways that improve its runtime behavior. Before the compiler can improve the code, it must locate points in the program where changing the code is likely to provide improvement, *and* it must prove that changing the code at those points is safe. Both of these tasks require a deeper understanding of the code than the compiler's front end typically derives. To gather the information needed to find opportunities for optimization and to justify those optimizations, compilers use some form of static analysis.

In general, static analysis involves compile-time reasoning about the runtime flow of values. This chapter explores techniques that compilers use to analyze programs in support of optimization.

Conceptual Roadmap

Compilers use static analysis to determine where optimizing transformations can be safely and profitably applied. In Chapter 8, we saw that optimizations operate on different scopes, from local to interprocedural. In

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00015-2

general, a transformation needs analytical information that covers at least as large a scope as the transformation; that is, a local optimization needs at least local information, while a whole-procedure, or global, optimization needs global information.

Static analysis generally begins with control-flow analysis; the compiler builds a graph that represents the flow of control within the code. Next, the compiler analyzes the details of how values flow through the code. It uses the resulting information to find opportunities for improvement and to prove the safety of transformations. Data-flow analysis was developed to answer these questions.

Static single-assignment (SSA) form is an intermediate representation that unifies the results of control-flow and data-flow analysis in a single sparse data structure. It has proven useful in both analysis and transformation and has become a standard IR used in both research and production compilers.

Overview

Chapter 8 introduced the subject of analysis and transformation of programs by examining local methods, regional methods, global methods, and interprocedural methods. Value numbering is algorithmically simple, even though it achieves complex effects; it finds redundant expressions, simplifies code based on algebraic identities and zero, and propagates known constant values. By contrast, finding an uninitialized variable is conceptually simple, but it requires the compiler to analyze the entire procedure to track definitions and uses.

Join point
In a CFG, a *join point* is a node that has multiple predecessors.

The difference in complexity between these two problems lies in the kinds of control flows that they encounter. Local and superlocal value numbering deal with subsets of the control-flow graph (CFG) that form trees (see Sections 8.4.1 and 8.5.1). To analyze the entire procedure, the compiler must reason about the full CFG, including cycles and *join points*, which both complicate analysis. In general, methods that only handle acyclic subsets of the CFG are amenable to online solutions, while those that deal with cycles in the CFG require offline solutions—the entire analysis must complete before rewriting can begin.

Static analysis
analysis performed at compile time or link time

Dynamic analysis
analysis performed at runtime, perhaps in a JIT or specialized self-modifying code

Static analysis, or compile-time analysis, is a collection of techniques that compilers use to prove the safety and profitability of a potential transformation. Static analysis over single blocks or trees of blocks is typically straightforward. This chapter focuses on global analysis, where the CFG can contain both cycles and join points. It mentions several problems in interprocedural analysis; these problems operate over the program's call graph or some related graph.

In simple cases, static analysis can produce precise results—the compiler can know exactly what will happen when the code executes. If the compiler can derive precise information, it might determine that the code evaluates to a known constant value and replace the runtime evaluation of an expression or function with an immediate load of the result. On the other hand, if the code reads values from any external source, involves even modest amounts of control flow, or encounters any ambiguous memory references, such as pointers, array references, or call-by-reference parameters, then static analysis becomes much harder and the results of the analysis are less precise.

This chapter begins with classic problems in data-flow analysis. We focus on an iterative algorithm for solving these problems because it is simple, robust, and easy to understand. Section 9.3 presents an algorithm for constructing SSA form for a procedure. The construction relies heavily on results from data-flow analysis. The advanced topics section explores the notion of flow-graph reducibility, presents a data structure that leads to a faster version of the dominator calculation, and provides an introduction to interprocedural data-flow analysis.

A Few Words About Time

The compiler analyzes the program to determine where it can safely apply transformations to improve the program. This *static analysis* either proves facts about the runtime flow of control and the runtime flow of values, or it approximates those facts. The analysis, however, takes place at compile time. In a classical ahead-of-time compiler, analysis occurs before any code runs.

Some systems employ compilation techniques at runtime, typically in the context of a just-in-time (JIT) compiler (see Chapter 14). With a JIT, the analysis and transformation both take place during runtime, so the cost of optimization counts against the program's runtime. Those costs are incurred on every execution of the program.

9.2 ITERATIVE DATA-FLOW ANALYSIS

Compilers use data-flow analysis, a set of techniques for compile-time reasoning about the runtime flow of values, to locate opportunities for optimization and to prove the safety of specific transformations. As we saw with live analysis in Section 8.6.1, problems in data-flow analysis take the form of a set of simultaneous equations defined over sets associated with the nodes and edges of a graph that represents the code being analyzed. Live analysis is formulated as a global data-flow problem that operates on the control-flow graph (CFG) of a procedure.

Forward problem
a problem in which the facts at a node *n* are computed based on the facts known for *n*'s CFG predecessors

Backward problem
a problem in which the facts at a node *n* are computed based on the facts known for *n*'s CFG successors.

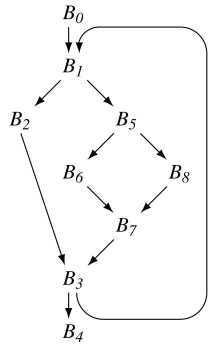
In this section, we will explore global data-flow problems and their solutions in greater depth. We will focus on one specific solution technique: an iterative fixed-point algorithm. It has the advantages of simplicity, speed, and robustness. We will first examine a simple forward data-flow problem, dominators in a flow graph. For a more complex example, we will return to the computation of LIVEOUT sets, a backward data-flow problem.

9.2.1 **Dominance**

Dominance
In a flow graph with entry node b_0, node b_i *dominates* node b_j, written $b_i \gg b_j$, if and only if b_i lies on all paths from b_0 to b_j. By definition, $b_i \gg b_i$.

Many optimization techniques must reason about the structural properties of the underlying code and its CFG. A key tool that compilers use to reason about the shape and structure of the CFG is the notion of *dominance*. Compilers use dominance to identify loops and to understand code placement. Dominance plays a key role in the construction of SSA form.

Many algorithms have been proposed to compute dominance information. This section presents a simple data-flow problem that annotates each CFG node b_i with a set DOM(b_i). A node's DOM set contains the names of all the nodes that dominate b_i.



Example CFG

n	DOM	IDOM
B_0	{0}	—
B_1	{0,1}	0
B_2	{0,1,2}	1
B_3	{0,1,3}	1
B_4	{0,1,3,4}	3
B_5	{0,1,5}	1
B_6	{0,1,5,6}	5
B_7	{0,1,5,7}	5
B_8	{0,1,5,8}	5

DOM and IDOM for the Example CFG

To make the notion of dominance concrete, consider node B_6 in the CFG shown in the margin. Every path from the entry node, B_0, to B_6 includes B_0, B_1, B_5, and B_6, so DOM(B_6) is $\{B_0, B_1, B_5, B_6\}$. The table in the margin shows all of the DOM sets for the CFG.

For any CFG node n, one $m \in$ DOM(n), $m \neq n$, will be closer to n in the CFG than any other $x \in$ DOM(n), $x \neq n$. That node, m, is the *immediate dominator* of n, denoted IDOM(n). By definition, a flow graph's entry node has no immediate dominator.

The following equations both define the DOM sets and form the basis of a method for computing them:

$$\text{DOM}(n) = \{n\} \cup \left(\bigcap_{m \in preds(n)} \text{DOM}(m) \right)$$

To provide initial values, the compiler sets:

$$\text{DOM}(n_0) = \{n_0\}$$
$$\text{DOM}(n) = N, \forall n \neq n_0$$

where N is the set of all nodes in the CFG. Given an arbitrary flow graph—that is, a directed graph with a single entry and a single exit—the equations

```
DOM(0) ← {0}
for i ← 1 to |N| - 1 do
    DOM(i) ← N

changed ← true
while (changed) do
    changed ← false

    for i ← 1 to |N| - 1 do
        temp ← {i} ∪ ( $\bigcap_{j \in preds(i)}$ DOM(j))

        if temp ≠ DOM(i) then
            DOM(i) ← temp
            changed ← true
```

■ **FIGURE 9.1** Iterative Solver for Dominance.

specify the DOM set for each node. At each join point in the CFG, the equations compute the intersection of the DOM sets along each entering path. Because they specify DOM(n) as a function of n's predecessors, denoted *preds*(n), information flows forward along edges in the CFG. Thus, the equations create a forward data-flow problem.

To solve the equations, the compiler can use the same three-step process used for live analysis in Section 8.6.1. It must (1) build a CFG, (2) gather initial information for each block, and (3) solve the equations to produce the DOM sets for each block. For DOM, step 2 is trivial; the computation only needs to know the node numbers.

Fig. 9.1 shows a round-robin iterative solver for the dominance equations. It considers the nodes in order by their CFG name, B_0, B_1, B_2, and so on. It initializes the DOM set for each node, then repeatedly recomputes those DOM sets until they stop changing.

Fig. 9.2 shows how the values in the DOM sets change as the computation proceeds. The first column shows the iteration number; iteration zero shows the initial values. Iteration one computes correct DOM sets for any node with a single path from B_0, but computes overly large DOM sets for B_3, B_4, and B_7. In iteration two, the smaller DOM set for B_7 corrects the set for B_3, which, in turn shrinks DOM(B_4). Similarly, the set for B_8 corrects the set for B_7. Iteration three shows that the algorithm has reached a fixed point.

Three critical questions arise regarding this solution procedure. First, does the algorithm halt? It iterates until the DOM sets stop changing, so the argument for termination is not obvious. Second, does it produce correct DOM sets? The answer is critical if we are to use DOM sets in optimizations. Finally, how fast is the solver? Compiler writers should avoid algorithms that are unnecessarily slow.

	Dom(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
0	{0}	N	N	N	N	N	N	N	N
1	{0}	{0,1}	{0,1,2}	{0,1,2,3}	{0,1,2,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,6,7}	{0,1,5,8}
2	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
3	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}

■ **FIGURE 9.2** Iterations in the Dominance Calculation.

Termination

Iterative calculation of the Dom sets halts because the sets that approximate Dom shrink monotonically throughout the computation. The algorithm initializes Dom(n_0) to {0}, and initializes the Dom sets for all other nodes to N, the set of all nodes. A Dom set can be no smaller than {0} and can be no larger than N. Careful reasoning about the while loop shows that a Dom set, say Dom(n_i), cannot grow from iteration to iteration. Either it shrinks, as the Dom set of one of its predecessors shrinks, or it remains unchanged.

The while loop halts when it makes a pass over the nodes in which no Dom set changes. Since the Dom sets can only change by shrinking and those sets are bounded in size, the while loop must eventually halt. When it halts, it has found a fixed point for this particular instance of the Dom computation.

Correctness

Recall the definition of dominance. Node n_i dominates n_j if and only if every path from the entry node n_0 to n_j contains n_i. Dominance is a property of paths in the CFG.

Dom(n_j) contains i if and only if $i \in$ Dom(n_k) for all $k \in preds(j)$, or if $i = j$. The algorithm computes Dom(n_j) as j plus the intersection of the Dom sets of all n_j's predecessors. How does this local computation over individual edges relate to the dominance property, which is defined over all paths through the CFG?

The Dom sets computed by the iterative algorithm form a fixed-point solution to the equations for dominance. The theory of iterative data-flow analysis, which is beyond the scope of this text, assures us that a fixed point exists for these particular equations and that the fixed point is unique [221]. This “all-paths” formulation of Dom describes a fixed-point for the equations, called the *meet-over-all-paths* solution. Uniqueness guarantees that the fixed point found by the iterative algorithm is identical to the meet-over-all-paths solution.

Meet operator

In the theory of data-flow analysis, the *meet operator* ($\wedge$) is used to combine facts at a join point in the CFG.

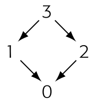
In the Dom equations, the meet operator is set intersection.

Efficiency

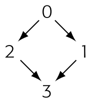
Because the fixed-point solution to the Dom equations for a specific CFG is unique, the solution is independent of the order in which the solver computes those sets. Thus, the compiler writer is free to choose an order of evaluation that improves the analyzer's running time.

Postorder number
a labeling of the graph's nodes that corresponds to the order in which a postorder traversal would visit them

A *reverse postorder* (RPO) traversal of the graph is particularly effective for forward data-flow problems. If we assume that the postorder numbers run from zero to |N| - 1, then a node's RPO number is simply |N| - 1 minus that node's postorder number. Here, N is the set of nodes in the graph.



Postorder



Reverse Postorder

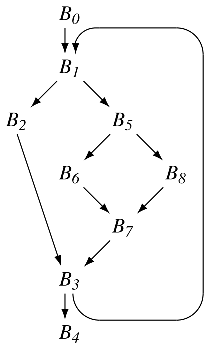
An RPO traversal visits as many of a node's predecessors as possible, in a consistent order, before visiting the node. (In a cyclic graph, a node's predecessor may also be its descendant.) A postorder traversal has the opposite property; for a node n, it visits as many of n's successors as possible before visiting n. Most interesting graphs will have multiple RPO numberings; from the perspective of the iterative algorithm, they are equivalent.

The compiler can compute RPO numbers in a postorder traversal if it starts a counter at |N| - 1 and decrements the counter as it visits and labels each node.

For a forward data-flow problem, such as Dom, the iterative algorithm should use an RPO computed on the CFG. For a backward data-flow problem, such as LiveOut, the algorithm should use an RPO computed on the *reverse* CFG; that is, the CFG with its edges reversed. (The compiler may need to add a unique exit node to ensure that the reverse CFG has a unique entry node.)

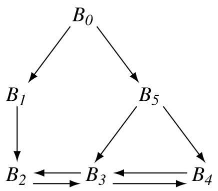
To see the impact of ordering, consider the impact of an RPO traversal on our example Dom computation. One RPO numbering for the example CFG, repeated in the margin, is:

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
RPO(n)	0	1	6	7	8	2	4	5	3



Example CFG

Visiting the nodes in this order produces the sequence of iterations and values shown in Fig. 9.3. Working in RPO, the algorithm computes accurate Dom sets for this graph on the first iteration and halts after the second iteration. RPO lets the algorithm halt in two passes over the graph rather than three. Note, however, that the algorithm does not always compute accurate Dom sets in the first pass, as the next example shows.



CFG with Double-Entry Loops

As a second example, consider the second CFG in the margin. It has two loops with multiple entries: (B_2,B_3) and (B_3,B_4). In particular, (B_2,B_3) has entries from both (B_0,B_1,B_2) and (B_0,B_5,B_3), while (B_3,B_4) has entries

	DOM(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
0	{0}	N	N	N	N	N	N	N	N
1	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
2	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}

■ **FIGURE 9.3** Iterations in the Reverse Postorder Dominance Calculation.

from (B_0,B_5,B_3) and (B_0,B_5,B_4). This property makes the graph irreducible, which makes it more difficult to analyze with some data-flow algorithms (see the discussion of reducibility in Section 9.5.1).

To apply the iterative algorithm, we need an RPO numbering. One RPO numbering for this CFG is:

	B_0	B_1	B_2	B_3	B_4	B_5
RPO(n)	0	2	3	4	5	1

Working in this order, the algorithm produces the following iterations:

	Dom(n)					
	B_0	B_1	B_2	B_3	B_4	B_5
0	{0}	N	N	N	N	N
1	{0}	{0,1}	{0,1,2}	{0,3}	{0,4}	{0,5}
2	{0}	{0,1}	{0,2}	{0,3}	{0,4}	{0,5}
3	{0}	{0,1}	{0,2}	{0,3}	{0,4}	{0,5}

The algorithm requires two iterations to compute the correct Dom sets. The final iteration recognizes that it has reached a fixed point.

The dominance calculation relies only on the structure of the graph. It ignores the behavior of the code in any of the CFG's blocks. As such, it might be considered a form of control-flow analysis. Most data-flow problems involve reasoning about the behavior of the code and the flow of data between operations. As an example of this kind of calculation, we will revisit the analysis of live variables.

9.2.2 Live-Variable Analysis

In Section 8.6.1, we used the results of live analysis to identify uninitialized variables. Compilers use live information for many other purposes, such

NAMING SETS IN DATA-FLOW EQUATIONS

In writing the data-flow equations for classic problems, we have renamed the sets that contain local information. The original papers use more intuitive set names. Unfortunately, those names clash with each other across problems. For example, *available expressions*, *live variables*, *reaching definitions*, and *anticipable expressions* all use some notion of a *kill* set. These four problems, however, are defined over three distinct domains: expressions (AVAILOUT and ANTOUT), definition points (REACHES), and variables (LIVEOUT). Thus, using one set name, such as KILL or KILLED, can produce confusion across problems.

The names that we have adopted encode both the domain and a hint as to the set's meaning. Thus, VARKILL(*n*) contains the set of variables killed in block *n*, while EXPRKILL(*n*) contains the set of expressions killed in the same block. Similarly, UEVAR(*n*) is the set of upward-exposed variables in *n*, while UEEXPR(*n*) is the set of upward-exposed expressions. While these names are somewhat awkward, they make explicit the distinction between the notion of kill used in *available expressions* (EXPRKILL) and the one used in *reaching definitions* (DEFKILL).

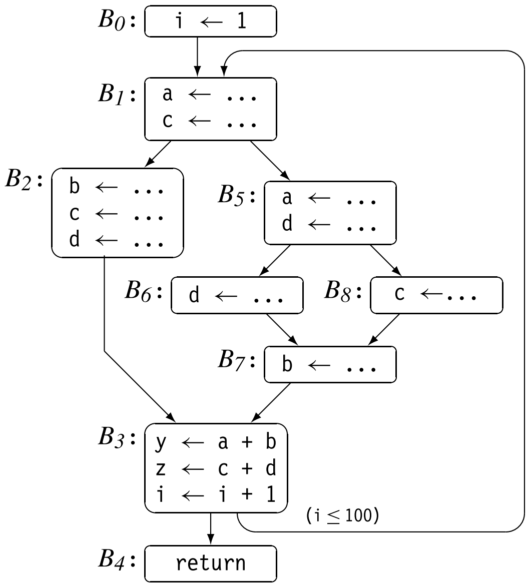
as register allocation and construction of some variants of SSA form. We formulated live analysis as a global data-flow problem with the equation:

$$\text{LiveOut}(n) = \bigcup_{m \in succ(n)} \left(\begin{array}{l} \text{UEVar}(m) \; \cup \\ (\text{LiveOut}(m) \; \cap \; \overline{\text{VarKill}(m)}\,) \end{array} \right)$$

where, *succ(n)* refers to the set of CFG successors of *n*. The analysis should initialize $\text{LiveOut}(n) = \emptyset, \; \forall n$.

Comparing the equations for LIVEOUT and DOM reveals differences between the problems.

- LIVEOUT is a backward data-flow problem; LIVEOUT(*n*) is a function of the information known on entry to each of *n*'s CFG successors. By contrast, DOM is a forward data-flow problem.
- LIVEOUT looks for a future use on *any path* in the CFG; thus, it combines information from multiple paths with the union operator. DOM looks for predecessors that lie on *all paths* from the entry node; thus, it combines information from multiple paths with the intersection operator.
- LIVEOUT reasons about the effects of operations. The sets UEVAR(*n*) and VARKILL(*n*) encode the effects of executing the block associated with *n*. By contrast, the DOM equations only use node names. LIVEOUT uses more information and takes more space.



(a) Code for Live Analysis Example

	UEVar	VarKill
B_0	∅	{i}
B_1	∅	{a,c}
B_2	∅	{b,c,d}
B_3	{a,b,c,d,i}	{y,z,i}
B_4	∅	∅
B_5	∅	{a,d}
B_6	∅	{d}
B_7	∅	{b}
B_8	∅	{c}

(b) Initial Sets for the Example

■ **FIGURE 9.4** Example Code for Live Analysis.

Despite the differences, the process for solving an instance of LiveOut is the same as for an instance of Dom. The compiler must: (1) build a CFG; (2) compute initial values for the sets (see Fig. 8.15(a) on page 420), and (3) apply the iterative algorithm (see Fig. 8.15(b)). These steps are analogous to those taken to solve the Dom equations.

To see the issues that arise in solving an instance of LiveOut, consider the code shown in Fig. 9.4(a). It fleshes out the example CFG that we have used throughout this chapter. Panel (b) shows the UEVar and VarKill sets for each block.

Fig. 9.5 shows the progress of the iterative solver on the example from Fig. 9.4(a), using the same RPO that we used in the Dom computation. Although the equations for LiveOut are more complex than those for Dom, the arguments for termination, correctness, and efficiency are similar to those for the dominance equations.

Termination

Recall that in Dom the sets shrink monotonically.

Iterative live analysis halts because the sets grow monotonically and the sets have a finite maximum size. Each time that the algorithm evaluates the

	LiveOut(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
0	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	∅	∅	{a,b,c,d,i}	∅	∅	∅	∅	{a,b,c,d,i}	∅
2	∅	{a,i}	{a,b,c,d,i}	{i}	∅	∅	{a,c,d,i}	{a,b,d,c,i}	{a,c,d,i}
3	{i}	{a,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
4	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
5	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}

■ **FIGURE 9.5** Iterations of the Live Solver Using the Order: $(B_0, B_1, B_5, B_8, B_6, B_7, B_2, B_3, B_4)$.

LiveOut equation at a node in the CFG, that LiveOut set either remains the same or it grows larger. The LiveOut sets do not shrink. When the algorithm reaches a state where no LiveOut set changes, it halts. It has reached a fixed point.

The LiveOut sets are finite. Each LiveOut set is either V, the set of names being analyzed, or it is a proper subset of V. In the worst case, one LiveOut set would grow by a single name in each iteration; that behavior would halt after $n \cdot |V|$ iterations, where n is the number of nodes in the CFG.

In the code from Fig. 9.4, V is {a, b, c, d, i, y, z}

This property—the combination of monotonicity and finite sets—guarantees termination. It is often called the *finite descending chain property*. In Dom, the sets shrink monotonically and their size is less than or equal to the number of nodes in the CFG. In LiveOut, the sets grow monotonically and their size is bounded by the number of names being analyzed. Either way, it guarantees termination.

Correctness

Iterative live analysis is correct if and only if it finds all the variables that satisfy the definition of liveness at the end of each block. Recall the definition: A variable v is *live* at point p if and only if there is a path from p to a use of v along which v is not redefined. Thus, liveness is defined in terms of paths in the CFG. A path that contains no definitions of v must exist from p to a use of v. We call such a path a v-clear path.

LiveOut(n) should contain v if and only if v is live at the end of block n. To form LiveOut(n), the iterative solver computes the contribution to LiveOut(n) of each successor of n in the CFG. The contribution of some successor m to LiveOut(n) is given by the right-hand side of the LiveOut equation: $\text{UEVar}(m) \cup (\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)})$. The

	LiveOut(n)								
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
0	∅	∅	∅	∅	∅	∅	∅	∅	∅
1	{i}	{a,c,i}	{a,b,c,d,i}	∅	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
2	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}
3	{i}	{a,c,i}	{a,b,c,d,i}	{i}	∅	{a,c,d,i}	{a,c,d,i}	{a,b,c,d,i}	{a,c,d,i}

■ **FIGURE 9.6** Iterations of the Live Solver Using RPO on the Reverse CFG.

solver combines the contributions of the various successors with union because $v \in$ LiveOut(n) if v is live on *any* path that leaves n.

How does this local computation over single edges relate to liveness defined over all paths? The LiveOut sets that the solver computes are a fixed-point solution to the live equations. Again, the theory of iterative data-flow analysis assures us that the live equations have a unique fixed-point solution [221]. Uniqueness guarantees that all the fixed-point solutions are identical, which includes the meet-over-all-paths solution implied by the definition.

Efficiency

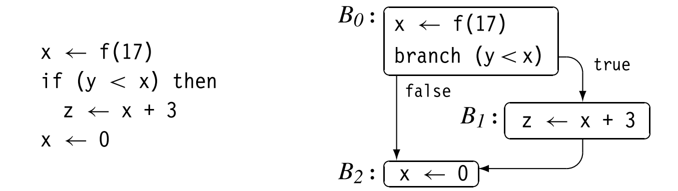
It is tempting to think that reverse postorder on the reverse CFG is equivalent to reverse preorder on the CFG. Exercise 3.b shows a counter-example.

For a backward problem, the solver should use an RPO traversal on the reverse CFG. The iterative evaluation shown in Fig. 9.5 used RPO on the CFG. For the example CFG, one RPO on the reverse CFG is:

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
RPO(n)	8	7	6	1	0	5	4	2	3

Visiting the nodes in this order produces the iterations shown in Fig. 9.6. Now, the algorithm halts in three iterations, rather than the five iterations required with a traversal ordered by RPO on the CFG. Comparing this table against the earlier computation, we can see why. On the first iteration, the algorithm computed correct LiveOut sets for all nodes except B_3. It took a second iteration for B_3 because of the back edge—the edge from B_3 to B_1. The third iteration is needed to recognize that the algorithm has reached its fixed point. Since the fixed point is unique, the compiler can use this more efficient order.

This pattern holds across many data-flow problems. The first iteration computes sets that are correct, except for the effects of cycles. Subsequent iterations settle out the information from cycles.



```
x ← f(17)
if (y < x) then
  z ← x + 3
x ← 0
```

(a) Simple If-Then Construct

(b) Corresponding Control-Flow Graph

■ **FIGURE 9.7** Control Flow Limits the Precision of Data-Flow Analysis.

9.2.3 **Limitations on Data-Flow Analysis**

There are limits to what a compiler can learn from data-flow analysis. In some cases, the limits arise from the assumptions underlying the analysis. In other cases, the limits arise from features of the language being analyzed. To make informed decisions, the compiler writer must understand what data-flow analysis can do and what it cannot do.

When it computes LIVEOUT(n), the iterative algorithm uses the sets LIVEOUT, UEVAR, and VARKILL for each of n's CFG successors. This action implicitly assumes that execution can reach each of those successors; in practice, one or more of them may not be reachable.

Consider the code fragment shown in Fig. 9.7 along with its CFG. The definition of x in B_0 is live on exit from B_0 because of the use of x in B_1. The definition of x in B_2 kills the value set in B_0. If B_1 cannot execute, then x's value from B_0 is not live past the comparison with y, and x $\notin$ LIVEOUT(B_0). If the compiler can prove that the y is always less than x, then B_1 never executes. The compiler can eliminate B_1 and replace the test and branch in B_0 with a jump to B_2. At that point, if the call to f has no side effects, the compiler can also eliminate B_0.

The equations for LIVEOUT, however, take the union over all successors of a block, not just its executable successors. Thus, the analyzer computes:

$$
\begin{aligned}
\text{LiveOut}(B_0) \;=\; & (\,\text{UEVar}(B_1) \cup (\text{LiveOut}(B_1) \cap \overline{\text{VarKill}(B_1)})\,) \\
\cup\; & (\,\text{UEVar}(B_2) \cup (\text{LiveOut}(B_2) \cap \overline{\text{VarKill}(B_2)})\,)
\end{aligned}
$$

Data-flow analysis assumes that all paths through the CFG are feasible. Thus, the information that they compute summarizes the possible data-flow events, assuming that each path can be taken. This limits the precision of the resulting information; we say that the information is precise "up to symbolic execution." With this assumption, x $\in$ LIVEOUT(B_0) and both B_0 and B_1 must be preserved.

STATIC ANALYSIS VERSUS DYNAMIC ANALYSIS

The notion of static analysis leads directly to the question: What about dynamic analysis? By definition, static analysis tries to estimate, at compile time, what will happen at runtime. In many situations, the compiler cannot tell what will happen, even though the answer might be obvious with knowledge of one or more runtime values.

Consider, for example, the C fragment:

```
x = y * z + 12;
*p = 0;
q = y * z + 13;
```

It contains a redundant expression, y * z, if and only if p does not contain the address of either y or z. At compile time, the value of p and the address of y and z may be unknown. At runtime, they are known and can be tested. Testing these values at runtime would allow the code to avoid recomputing y * z, where compile-time analysis might be unable to answer the question.

However, the cost of testing whether p == &y, or p == &z, or neither and acting on the result is likely to exceed the cost of recomputing y * z. For dynamic analysis to make sense, it must be a priori profitable—that is, the savings must exceed the cost of the analysis. This happens in some cases; in most cases, it does not. By contrast, the cost of static analysis can be amortized over multiple executions of the code, so it is more attractive, in general.

Another way that imprecision creeps into the results of data-flow analysis comes from the treatment of arrays, pointers, and procedure calls. An array reference, such as A[i,j], refers to a single element of A. However, without analysis that reveals the values of i and j, the compiler cannot tell which element of A is accessed. For this reason, compilers have traditionally treated a reference to an element of A as a reference to all of A. Thus, a use of A[i,j] counts as a use of A, and a definition of A[m,n] counts as a definition of A.

The compiler writer must not, however, make too strong an inference. Because the information on arrays is imprecise, the compiler must interpret that information conservatively. Thus, if the goal of the analysis is to determine where a value is no longer live—that is, the value *must have been killed*—then a definition of A[i,j] does not kill the value of A. If the goal is to recognize where a value *might not survive*, then a definition of A[i,j] *might* define any element of A.

Pointers add another level of imprecision to the results of static analysis. Explicit arithmetic on pointers makes matters worse. Unless the compiler employs an analysis that tracks the values of pointers, it must interpret an assignment to a pointer-based variable as a potential definition for every variable that the pointer might reach. Type safety can limit the set of objects that the pointer can define; a pointer declared to point at an object of type *t* can only be used to modify objects of type *t*. Without analysis of pointer values or a guarantee of type safety, assignment to a pointer-based variable can force the analyzer to assume that every variable has been modified. In practice, this effect often prevents the compiler from keeping the value of a pointer-based variable in a register across any pointer-based assignment. Unless the compiler can specifically prove that the pointer used in the assignment cannot refer to the memory location corresponding to the enregistered value, it cannot safely keep the value in a register.

Points-to analysis, used to track possible pointer values, is more expensive than classic data-flow problems such as DOM and LIVE.

The complexity of analyzing pointer use leads many compilers to avoid keeping values in registers if they can be the target of a pointer. Usually, some variables can be exempted from this treatment—such as a local variable whose address has never been explicitly taken. The alternative is to perform data-flow analysis aimed at disambiguating pointer-based references—reducing the set of possible variables that a pointer might reference at each point in the code. If the program can pass pointers as parameters or use them as global variables, pointer disambiguation becomes inherently interprocedural.

Procedure calls provide a final source of imprecision. To understand the data flow in the current procedure, the compiler must know what the callee can do to each variable that is accessible to both the caller and the callee. The callee may, in turn, call other procedures that have their own potential side effects.

Unless the compiler computes accurate summary information for each procedure call, it must estimate the call's worst-case behavior. While the specific assumptions vary across problems and languages, the general rule is to assume that the callee both uses and modifies every variable that it can reach. Since few procedures modify and use every variable, this rule typically overestimates the impact of a call, which introduces further imprecision into the results of the analysis.

9.2.4 **Other Data-Flow Problems**

Compilers use data-flow analyses to prove the safety of applying transformations in specific situations. Thus, many distinct data-flow problems have been proposed, each for a particular optimization.

Available Expressions

Availability
An expression e is *available* at point p if and only if, on every path from the procedure's entry to p, e is evaluated and none of its operands is redefined.

To identify redundant expressions, the compiler can compute information about the *availability* of expressions. This analysis annotates each node n in the CFG with a set AVAILIN(n), which contains the names of all expressions in the procedure that are available on entry to the block corresponding to n. The equations for AVAILIN are:

$$\text{AVAILIN}(n) = \bigcap_{m \in preds(n)} \left(\begin{array}{l} \text{DEEXPR}(m) \;\cup \\ (\text{AVAILIN}(m) \;\cap\; \overline{\text{EXPRKILL}(m)}\,) \end{array} \right)$$

with initial values for the AVAILIN sets:

$$\begin{array}{lcl} \text{AVAILIN}(n_0) & = & \emptyset \\ \text{AVAILIN}(n) & = & \{\, \textit{all expressions} \;\}, \forall\, n \neq n_0 \end{array}$$

These equations can be solved efficiently with a standard iterative data-flow solver. Since it is a forward data-flow problem, the solver should use RPO on the CFG.

In the equations, DEEXPR(n) is the set of downward-exposed expressions in n. An expression $e \in$ DEEXPR(n) if and only if block n evaluates e and none of e's operands is defined between the last evaluation of e in n and the end of n. EXPRKILL(n) contains all those expressions that are killed by a definition in n. An expression is killed if one or more of its operands are redefined in the block.

An expression e is available on entry to n if and only if it is available on exit from each of n's predecessors in the CFG. As the equation states, an expression e is available on exit from some block m if one of two conditions holds: either e is downward exposed in m, or it is available on entry to m and is not killed in m.

AVAILIN sets are used in global redundancy elimination, sometimes called *global common subexpression elimination*. Perhaps the simplest way to achieve this effect is to compute AVAILIN sets for each block and use them as initial information in local value numbering (see Section 8.4.1). Lazy code motion is a stronger form of redundancy elimination that also uses availability (see Section 10.3.1).

Reaching Definitions

In some cases, the compiler needs to know where an operand was defined. If multiple paths in the CFG lead to the operation, then multiple definitions may provide the value of the operand. To find the set of definitions that reach a block, the compiler can compute *reaching definitions*. The compiler annotates each node n in the CFG with a set, REACHES(n) that contains the name of every definition that reaches the head of the block corresponding to n. The domain of REACHES is the set of definition points in the procedure—the set of assignments.

Reachability
A definition d of variable x *reaches* operation u if and only if u uses the value of x and there exists a path from d to u along which x is not redefined.

The compiler computes a set REACHES(n) for each CFG node n using the equation:

$$\text{Reaches}(n) = \bigcup_{m \in preds(n)} \left(\begin{array}{l} \text{DEDef}(m) \;\cup \\ (\text{Reaches}(m) \cap \overline{\text{DefKill}(m)}\,) \end{array} \right)$$

with initial values for the REACHES sets:

$$\text{Reaches}(n) = \emptyset, \forall\, n$$

DEDEF(m) is the set of downward-exposed definitions in m: those definitions in m for which the defined name is not subsequently redefined in m. DEFKILL(m) contains *all* the definition points that are obscured by a definition of the same name in m; $d \in$ DEFKILL(m) if d defines some name v and m contains a definition that also defines v. Thus, $\overline{\text{DefKill}(m)}$ contains those definition points that survive through m.

DEDEF and DEFKILL are both defined over the set of definition points, but computing each of them requires a mapping from names (variables and compiler-generated temporaries) to definition points. Thus, gathering the initial information for reaching definitions is more expensive than it is for live variables.

Anticipable Expressions

In some situations, the compiler can move an expression backward in the CFG and replace multiple instances of the expression, along different paths, with a single instance. This optimization, called *hoisting*, reduces code size. It does not change the number of times the expression is evaluated.

Anticipability
An expression, e, is *anticipable* at point p if and only if (1) every path that leaves p evaluates e, and (2) evaluating e at p would produce the same result as the first evaluation along each of those paths.

To find safe opportunities for hoisting, the compiler can compute the set of *anticipable expressions* at the end of each block. An expression e is anticipable at the end of block b if the next evaluation of e, along each path leaving b, would produce the same result. The equations require that e be computed along every path that leaves b.

ANTOUT, the set of expressions anticipable at the end of a block, can be computed as a backward data-flow problem on the CFG. Anticipability is formulated over the domain of expressions.

IMPLEMENTING DATA-FLOW FRAMEWORKS

The equations for many global data-flow problems show a striking similarity. For example, available expressions, live variables, reaching definitions, and anticipable expressions all have propagation functions of the form:

$$f(x) \;=\; c_1 \;\; op_1 \; (x \;\; op_2 \;\; c_2)$$

where c_1 and c_2 are constants derived from the code and op_1 and op_2 are standard set operations such as ∪ and ∩. This similarity appears in the problem descriptions; it creates the opportunity for code sharing in the implementation of the analyzer.

The compiler writer can easily abstract away the details in which these problems differ and implement a single, parameterized analyzer. The analyzer needs functions to compute c_1 and c_2, implementations of the operators, and an indication of the problem's direction. In return, it produces the desired data-flow sets.

This implementation strategy encourages code reuse. It hides the low-level details of the solver. It also creates a situation in which the compiler writer can profitably invest effort in optimizing the implementation. For example, a scheme that implements $f(x) = c_1 \; op_1 \; (x \; op_2 \; c_2)$ as a single function may outperform one that implements both $f_1(x) = c_1 \; op_1 \; x$ and $f_2(x) = x \; op_2 \; c_2$, and computes $f(x)$ as $f_1(f_2(x))$. A framework lets all the client transformations benefit from improvements in the set representations and operator implementations.

The equations to define AntOut are:

$$\text{AntOut}(n) \;=\; \bigcap_{m \in succ(n)} \left(\begin{array}{l} \text{UEExpr}(m) \;\cup \\ (\text{AntOut}(m) \cap \overline{\text{ExprKill}(m)}\,) \end{array} \right)$$

with initial values for the AntOut sets:

$$\begin{aligned} \text{AntOut}(n_f) &= \emptyset \\ \text{AntOut}(n) &= \{\, \textit{all expressions} \,\},\ \forall\, n \neq n_f \end{aligned}$$

Here UEExpr(m) is the set of upward-exposed expressions—those used in m before they are killed. ExprKill(m) contains all those expressions that are killed by a definition in m; it also appears in the equations for available expressions.

The results of anticipability analysis are used in lazy code motion, to decrease execution time, and in code hoisting, to shrink the size of the compiled code. Both transformations are discussed in Section 10.3.

Interprocedural Summary Problems

When analyzing a single procedure, the compiler must account for the impact of each procedure call. In the absence of specific information about the call, the compiler must make worst-case assumptions about the callee and about any procedures that it, in turn, calls. These assumptions can seriously degrade the precision of the global data-flow information. For example, the compiler must assume that the callee modifies every variable that it can access; this assumption essentially stops the propagation of facts across a call site for all global variables, module-level variables, and call-by-reference parameters.

To limit such impact, the compiler can compute summary information on each call site. The classic summary problems compute the set of variables that might be modified as a result of the call and that might be used as a result of the call. The compiler can then use these computed summary sets in place of its worst case assumptions.

The *interprocedural may modify problem* annotates each call site with a set of names that the callee, and procedures it calls, might modify. May modify is one of the simplest problems in interprocedural analysis, but it can have a significant impact on the quality of information produced by other analyses, such as global constant propagation. May modify is posed as a set of data-flow equations over the program's call graph that annotate each procedure with a MAYMOD set.

Flow insensitive
This formulation of MAYMOD ignores control flow inside procedures. Such a formulation is said to be *flow insensitive*.

$$\begin{aligned}\text{MAYMOD}(p) \;=\; & \text{LOCALMOD}(p) \;\cup \\ & (\cup_{e=(p,q)}\; \mathit{unbind}_e(\text{MAYMOD}(q)))\end{aligned}$$

MAYMOD(p) is initialized to contain all the names modified locally in p that are visible outside p. It is computed as the set of names defined in p minus any names that are strictly local to p.

The function unbind_e maps one set of names into another. For a call-graph edge $e = (p,q)$ and set of names s, $\mathit{unbind}_e(s)$ maps each name in s from the name space of q to the name space that holds at the call site, using the bindings at the call site that corresponds to e. In essence, it projects s from q's name space into p's name space.

Given a set of LOCALMOD sets and a call graph, an iterative solver will find a fixed-point solution for these equations. It will not achieve the kind of fast time bound seen in global data-flow analysis. A more complex framework is required to achieve near-linear complexity on this problem (see Chapter Notes).

The MayMod sets computed by these equations are generalized summary sets. That is, MayMod(q) contains the names of variables that might be modified by a call to q, expressed in the name space of q. To use this information at a specific call site that invokes q, the compiler will compute the set $S = \mathit{unbind}_e(\text{MayMod}(q))$, where $e = (p,q)$ is the call graph edge corresponding to the call. The compiler must then add to S any names that are aliased inside p to names contained in S.

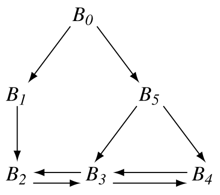
The compiler can also compute the set of variables that might be referenced as a result of executing a procedure call, the *interprocedural may reference problem*. The equations to annotate each procedure p with a set MayRef(p) are similar to the equations for MayMod.

SECTION REVIEW

Iterative data-flow analysis works by repeatedly reevaluating an equation at each node in some underlying graph until the sets defined by the equations reach a fixed point. Many data-flow problems have a unique fixed point, which ensures a correct solution independent of the evaluation order, and the finite descending chain property, which guarantees termination independent of the evaluation order. These two properties allow the compiler writer to choose evaluation orders that converge quickly. As a result, iterative analysis is robust and efficient.

The literature describes many different data-flow problems. Examples in this section include dominance, live analysis, availability, anticipability, and interprocedural summary problems. All of these, save for the interprocedural problems, have straightforward efficient solutions with the iterative algorithm. To avoid solving multiple problems, compilers often turn to a unifying framework, such as SSA form, described in the next section.

REVIEW QUESTIONS



1. Compute Dom sets for the CFG shown in the margin, evaluating the nodes in the order $\{B_4, B_2, B_1, B_5, B_3, B_0\}$. Explain why this order takes a different number of iterations than is shown on page 456.
2. When the compiler builds a call graph, ambiguous calls can complicate the process, much as ambiguous jumps complicate CFG construction. What language features might lead to an ambiguous call site—one where the compiler was uncertain of the callee's identify?

9.3 STATIC SINGLE-ASSIGNMENT FORM

Over time, compiler writers have formulated many different data-flow problems. If each transformation uses its own analysis, the effort spent implementing, debugging, and maintaining the analysis passes can grow unreasonably large. To limit the number of analyses that the compiler writer must implement and that the compiler must run, it is desirable to use a single analysis for multiple transformations.

One strategy for such a "universal" analysis is to build an IR called static single-assignment form (SSA) (see also Section 4.6.2). SSA encodes both data flow and control flow directly into the IR. Many of the classic scalar optimizations have been reworked to operate on code in SSA form.

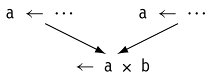
Some compilers, such as LLVM/CLANG, use SSA as their definitive IR.

Code in SSA form obeys two rules:

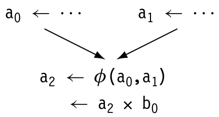
1. Each computation in the procedure defines a unique name.
2. Each use in the procedure refers to a single name.

The first rule removes the effect of "kills" from the code; any expression in the code is available at any point after it has been evaluated. (We first saw this effect in local value numbering.) The second rule has a more subtle effect. It ensures that the compiler can still represent the code concisely and correctly; a use can be written with a single name rather than a long list of all the definitions that might reach it.

Consider the small example shown in the margin. If the compiler renames the two definitions of a to a_0 and a_1, what name should appear in the use of a in $a \times b$? Neither a_0 nor a_1 will work in $a \times b$. (The example assumes that b was defined earlier in the code.)



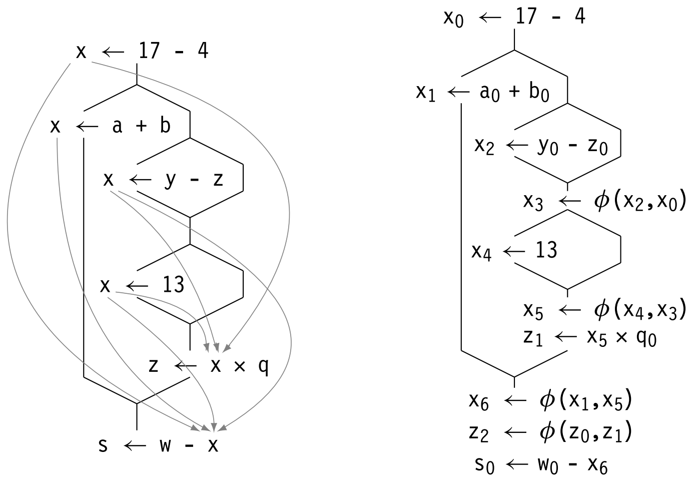
Original Code



Code in SSA Form

To manage this name space, the SSA construction inserts a special kind of copy operation, a ϕ-function, at the head of the block where control-flow paths meet, as shown in the margin. When the ϕ-function evaluates, it reads the argument that corresponds to the edge from which control flow entered the block. Thus, coming from the block on the left, the ϕ-function reads a_0, while from the block on the right it reads a_1. The selected argument is assigned to a_2. Thus, the evaluation of $a_2 \times b_0$ computes the same value that $a \times b$ did in the pre-SSA code.

Fig. 9.8 shows a more extensive example. Consider the various uses of the variable x in the code fragment shown in panel (a). The curved gray lines show which definitions can reach each use of x. Panel (b) shows the same fragment in SSA form. Variables have been renamed with subscripts to ensure unique names for each definition. We assume that a_0, b_0, w_0, y_0, and z_0 are defined earlier in the code.


x ← 17 - 4
x ← a + b
x ← y - z
x ← 13
z ← x × q
s ← w - x
$x_0 \leftarrow 17 - 4$
$x_1 \leftarrow a_0 + b_0$
$x_2 \leftarrow y_0 - z_0$
$x_3 \leftarrow \phi(x_2, x_0)$
$x_4 \leftarrow 13$
$x_5 \leftarrow \phi(x_4, x_3)$
$z_1 \leftarrow x_5 \times q_0$
$x_6 \leftarrow \phi(x_1, x_5)$
$z_2 \leftarrow \phi(z_0, z_1)$
$s_0 \leftarrow w_0 - x_6$

(a) Original Code Fragment

(b) With x and z in SSA Form

■ **FIGURE 9.8** SSA: Encoding Control Flow into Data Flow.

The code in panel (b) includes all of the ϕ-functions needed to reconcile the names generated by rule one with the need for unique names in uses. Tracing the flow of values will reveal that the same values follow the same paths as in the original code.

Two final points about ϕ-functions need explanation. First, ϕ-functions are defined to execute concurrently. When control enters a block, all of the block's ϕ-functions read their designated argument, in parallel. Next, they all define their target names, in parallel. This concurrent execution semantics allows the SSA construction algorithm to ignore the order of ϕ-functions as it inserts them into a block.

Second, by convention, we write the arguments of a ϕ-function left-to-right to correspond with the incoming edges left-to-right on the printed page. Inside the compiler, the IR has no natural notion of left-to-right for the edges entering a block. Thus, the implementation will require some bookkeeping to track the correspondence between ϕ-function arguments and CFG edges.

9.3.1 A Naive Method for Building SSA Form

Both of the SSA-construction algorithms that we present follow the same basic outline: (1) insert ϕ-functions as needed and (2) rename variables and temporary values to conform with the two rules that define SSA form. The simplest construction method implements the two steps as follows:

1. *Inserting ϕ-functions* At the start of each block that has multiple CFG predecessors, insert a ϕ-function, such as $x \leftarrow \phi(x,x)$, for each name x that the current procedure defines. The ϕ-function should have one argument for each predecessor block in the CFG. This process inserts a ϕ-function in every case that might need one. It also inserts many extraneous ϕ-functions.

The "naive" algorithm inserts more ϕ-functions than are needed. It adds a ϕ-function for each name at each join point.

2. *Renaming* The ϕ-function insertion algorithm ensures that a ϕ-function for x is in place at each join point in the CFG reached by two or more definitions of x. The renaming algorithm rewrites all of the names into the appropriate SSA names. The first step adds a unique subscript to the name at each definition.

   At this point, each definition has a unique SSA name. The compiler can compute reaching definitions (see Section 9.2.4) to determine which SSA name reaches each use. The compiler writer must change the meaning of DEFKILL so that a definition to one SSA name kills not only that SSA name but also all SSA names with the same base name. The effect is to stop propagation of an SSA name at any ϕ-function where it is an argument. With this change, exactly one definition—one SSA name—reaches each use.

   The compiler makes a pass over the code to rewrite the name in each use with the SSA name that reaches it. This process rewrites all the uses, including those in ϕ-function arguments. If the same SSA name reaches a ϕ-function along multiple paths, the corresponding ϕ-function arguments will have the same SSA name.

   The compiler must sort out the correspondence between incoming edges in the CFG and ϕ-function arguments so that it can rename each argument with the correct SSA name. While conceptually simple, this task requires some bookkeeping.

Base name
In an SSA name x_2, the base name is x and the version is 2.

The naive algorithm constructs SSA form that obeys the two rules. Each definition assigns to a unique name; each reference uses the name of a distinct definition. While the algorithm builds correct SSA form, it can insert ϕ-functions that are redundant or dead. These extra ϕ-functions may be problematic. The compiler wastes memory representing them and time traversing them. They can also decrease the precision of some kinds of analysis over SSA form.

A ϕ-function $x_j \leftarrow \phi(x_i, x_i)$ is redundant.

A ϕ-function whose value is not live is considered dead.

We call this flavor of SSA *maximal SSA form*. To build SSA form with fewer ϕ-functions requires more work; in particular, the compiler must analyze the code to determine where potentially distinct values converge in the CFG. This computation relies on the dominance information described in Section 9.2.1.

THE DIFFERENT FLAVORS OF SSA FORM

The literature proposes several distinct flavors of SSA form. The flavors differ in their criteria for inserting ϕ-functions. For a given program, they can produce different sets of ϕ-functions.

Minimal SSA inserts a ϕ-function at any join point where two distinct definitions for the same original name meet. This is the minimal number consistent with the definition of SSA. Some of those ϕ-functions, however, may be dead; the definition says nothing about the values being live when they meet.

Pruned SSA adds a liveness test to the ϕ-insertion algorithm to avoid adding dead ϕ-functions. The construction must compute LIVEOUT sets, which increases the cost of building pruned SSA.

Semipruned SSA is a compromise between minimal SSA and pruned SSA. Before inserting ϕ-functions, the algorithm eliminates any names that are not live across a block boundary. This can shrink the name space and reduce the number of ϕ-functions without the overhead of computing LIVEOUT sets. The algorithm in Fig. 9.11 computes semipruned SSA.

Of course, the number of ϕ-functions depends on the specific program being converted into SSA form. For some programs, the reductions obtained by semipruned SSA and pruned SSA are significant. Shrinking the SSA form can lead to faster compilation, since passes that use SSA form then operate on programs that contain fewer operations—and fewer ϕ-functions.

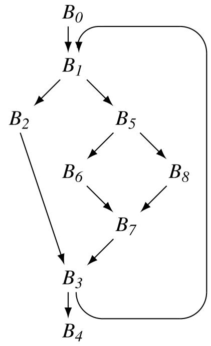
The following subsections present, in detail, an algorithm to build *semipruned SSA*—a version with fewer ϕ-functions. Section 9.3.2 introduces *dominance frontiers* and shows how to compute them; dominance frontiers guide ϕ-function insertion. Section 9.3.3 gives an algorithm to insert ϕ-functions, and Section 9.3.4 presents an efficient algorithm for renaming. Section 9.3.5 discusses complications that can arise in translating out of SSA form.

9.3.2 **Dominance Frontiers**

The primary problem with maximal SSA form is that it contains too many ϕ-functions. To reduce their number, the compiler must determine more carefully where they are needed. The key to ϕ-function insertion lies in understanding which names need a ϕ-function at each join point. To solve this problem efficiently and effectively, the compiler can turn the question around. It can determine, for each block i, the set of blocks that will need a ϕ-function as the result of a definition in block i. Dominance plays a critical role in this computation.

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DOM	{0}	{0,1}	{0,1,2}	{0,1,3}	{0,1,3,4}	{0,1,5}	{0,1,5,6}	{0,1,5,7}	{0,1,5,8}
IDOM	—	0	1	1	3	1	5	5	5
DF	∅	1	3	1	∅	3	7	3	7

■ **FIGURE 9.9** DOM, IDOM, and DF Sets for the Example CFG.



Example CFG

Consider the CFG shown in the margin. Assume that the code assigns distinct values to a in both B_1 and B_5, and that no other block assigns to a. The value from B_5 is the only value for a that can reach B_6, B_7, and B_8. Because B_5 dominates these three blocks, it lies on any path from B_0 to B_6, B_7, or B_8. The definition in B_1 cannot reach them.

B_3 presents a different situation. Neither of its CFG predecessors, B_2 and B_7, dominate B_3. A use of a in B_3 can receive its value from either B_1 or B_5, depending on the path taken to reach B_3. The assignments to a in B_1 and B_5 force a ϕ-function for a at the start of B_3.

B_5 dominates the region (B_6, B_7, B_8). It is the immediate dominator of all three nodes. A definition of a in B_5 will reach a use in that region, unless a is redefined before the use. The definition in B_5 cannot necessitate the need for a ϕ-function in this region.

B_3 lies just outside of the region that B_5 dominates. It has two CFG predecessors and B_5 only dominates one of them. Thus, it lies one CFG edge outside the region that B_5 dominates. In general, a definition of a in some block B_i will necessitate a ϕ-function in any node that, like B_3, lies one CFG edge beyond the region that B_i dominates. The *dominance frontier* of B_i, denoted DF(B_i), is the set of all such nodes.

Strict dominance

In a CFG, node p *strictly dominates* node q if $p \in$ DOM(q) and $p \neq q$.

We denote this as $p \in$ (DOM(q) $-$ q).

Dominance frontier

In a CFG, node q is in the *dominance frontier* of node p if and only if (1) p dominates a CFG predecessor of q and (2) p does not strictly dominate q.

We denote p's dominance frontier as DF(p).

To recap, $q \in$ DF(p) if, along some path, q is one edge beyond the region that p dominates. Thus:

- q has a CFG predecessor that p dominates. There exists an x such that (x, q) is a CFG edge and $p \in$ DOM(x).
- p does not strictly dominate q. That is, $p \notin$ (DOM(q) $-$ q).

DF(p) is simply the set of all nodes that meet these two criteria.

A definition of a in block n forces the insertion of a ϕ-function for a at the head of each block $m \in$ DF(n). Fig. 9.9 shows the DOM, IDOM, and DF sets for the example CFG.

Notice the role of strict dominance. In the example CFG, strict dominance ensures that $B_1 \in$ DF(B_1). Thus, an assignment to some name a in B_1 forces

```
for all nodes, n, in the CFG do
    $DF(n) \leftarrow \emptyset$
for all nodes, n, in the CFG do
    if n has multiple predecessors then
        for each CFG predecessor p of n do
            $runner \leftarrow p$
            while $runner \neq \text{IDom}(n)$ do
                $DF(runner) \leftarrow DF(runner) \cup \{n\}$
                $runner \leftarrow \text{IDom}(runner)$
```

■ **FIGURE 9.10** Algorithm for Computing Dominance Frontiers.

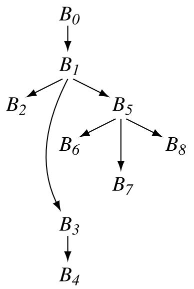
the insertion of a ϕ-function in B_1. If the definition of dominance frontiers used DOM, instead, DF(B_1) would be empty.

Dominator Trees

Dominator tree
a tree that encodes the dominance information for a flow graph

The algorithm to compute dominance frontiers uses a data structure, the *dominator tree*, to encode dominance relationships. The dominator tree of a CFG has a node for each block in the CFG. Edges encode immediate dominance; if $m = \text{IDom}(n)$, then n is a child of m in the dominator tree.

The dominator tree encodes the DOM sets as well. For a node n, DOM(n) contains precisely the nodes on the path from n to the root of the dominator tree. The nodes on that path are ordered by the IDOM relationship. The dominator tree for our running example appears in the margin.



Dominator Tree for the Example CFG

Computing Dominance Frontiers

To make ϕ-insertion efficient, the compiler should precompute, for each CFG node n, a set DF(n) that contains n's dominance frontier. The algorithm, shown in Fig. 9.10, uses both the dominator tree and the CFG to build the DF(n) sets.

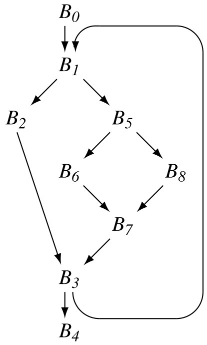
Notice that the DF sets can only contain nodes that are join points in the CFG—that is, nodes that have multiple predecessors. Thus, the algorithm starts with the join points. At a CFG join point n, it iterates over n's CFG predecessors p and inserts n into DF(p) as needed.

If $p = \text{IDom}(n)$, then p also dominates all of n's other predecessors. In the example, $B_0 = \text{IDom}(B_1)$.

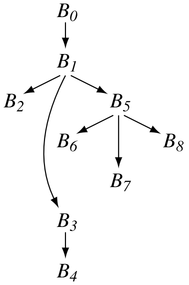
- If $p = \text{IDom}(n)$, then n does not belong to DF(p). Neither does it belong to DF(m) for any predecessor m of p.
- If $p \neq \text{IDom}(n)$, then n belongs in DF(p). It also belongs in DF(q) for any q such that $q \in \text{Dom}(p)$ and $q \notin (\text{Dom}(n) - n)$. The algorithm finds these latter nodes q by running up the dominator tree.

The algorithm follows from these observations. It initializes DF(n) to $\emptyset$, for all CFG nodes n. Next, it finds each CFG join point n and iterates over n's

CFG predecessors, p. If p dominates n, the algorithm is done with p. If not, it adds n to DF(p) and walks up the dominator tree, adding n to the DF set of each dominator-tree ancestor until it finds n's immediate dominator. The algorithm needs a small amount of bookkeeping to avoid adding n to a DF set multiple times.



Example CFG



Example Dominator Tree

Consider again the example CFG and its dominator tree. The analyzer examines the nodes in some order, looking for nodes with multiple predecessors. Assuming that it takes the nodes in name order, it finds the join points as B_1, then B_3, then B_7.

B_1 For CFG-predecessor B_0, the algorithm finds that B_0 is IDOM(B_1), so it never enters the while loop. For CFG-predecessor B_3, it adds B_1 to DF(B_3) and sets *runner* to IDOM(B_3) = B_1. It adds B_1 to DF(B_1) and sets *runner* to IDOM(B_1) = B_0, where it halts.

B_3 For CFG-predecessor B_2, it adds B_3 to DF(B_2) and sets *runner* to IDOM(B_2) = B_1. Since B_1 = IDOM(B_3), it halts. For CFG-predecessor B_7, it adds B_3 to DF(B_7) and sets *runner* to IDOM(B_7) = B_5. It adds B_3 to DF(B_5) and sets *runner* to IDOM(B_5) = B_1, where it halts.

B_7 For CFG-predecessor B_6, it adds B_7 to DF(B_6) and advances *runner* to IDOM(B_6) = B_5, where it halts. For CFG-predecessor B_8, it adds B_7 to DF(B_8) and advances *runner* to IDOM(B_8) = B_5, where it halts.

These results produce the DF sets shown in the table in Fig. 9.9.

9.3.3 Placing ϕ-Functions

The naive algorithm placed a ϕ-function for every variable at the start of every join node. With dominance frontiers, the compiler can determine more precisely where ϕ-functions might be needed. The basic idea is simple.

- From a control-flow perspective, an assignment to x in CFG node n induces a ϕ-function in every CFG node $m \in$ DF(n). Each inserted ϕ-function creates a new assignment; that assignment may, in turn, induce additional ϕ-functions.
- From a data-flow perspective, a ϕ-function is only necessary if its result is live at the point of insertion. The compiler could compute live information and check each ϕ-function on insertion; that approach leads to *pruned SSA form.*

In practice, the compiler can avoid most dead ϕ-functions with an inexpensive approximation to liveness. A name x cannot need a ϕ-function unless it is live in multiple blocks. The compiler can compute the set of *global names*—those that are live in multiple blocks. The SSA-construction can

The word *global* is used here to mean of interest across the entire procedure.

```
Globals ← ∅
Initialize all the Blocks sets to ∅
for each block b
    VarKill ← ∅
    for each operation i in b, in order
        assume that $op_i$ is “x ← y op z”
        if y ∉ VarKill then
            Globals ← Globals ∪ {y}
        if z ∉ VarKill then
            Globals ← Globals ∪ {z}
        VarKill ← VarKill ∪ {x}
        Blocks(x) ← Blocks(x) ∪ {b}
```

(a) Finding Global Names

```
for each name x ∈ Globals
    WorkList ← Blocks(x)
    for each block b ∈ WorkList
        remove b from WorkList
        for each block d in DF(b)
            if d has no ϕ-function for x then
                insert a ϕ-function for x in d
                WorkList ← WorkList ∪ {d}
```

(b) Inserting ϕ-Functions

■ **FIGURE 9.11** ϕ-Function Insertion.

ignore any nonglobal name, which reduces the name space and the number of ϕ-functions. The resulting SSA form is called *semipruned SSA form*.

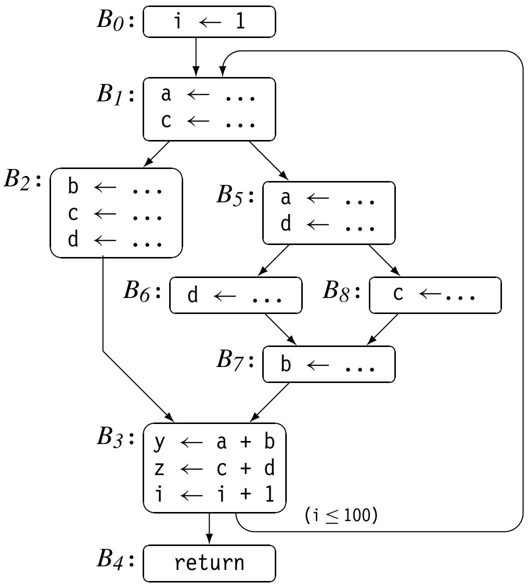
The compiler can find the global names cheaply. In each block, it looks for names with upward-exposed uses—the UEVAR set from the live-variables calculation. Any name that appears in a LIVEOUT set must be in the UEVAR set of some block. Taking the union of all the UEVAR sets gives the compiler the set of names that are live on entry to one or more blocks and, hence, live in multiple blocks.

The algorithm to find global names, shown in Fig. 9.11(a), is derived from the obvious algorithm for computing UEVAR. It constructs both a set of global names, *Globals*, and, for each name, the set of blocks that contain a definition of that name. The algorithm uses these block lists to form initial worklists during ϕ-function insertion.

The algorithm for inserting ϕ-functions, in panel (b), iterates over the global names. For each name *x*, it initializes *WorkList* with *Blocks(x)*. For each block *b* in *WorkList*, it inserts a ϕ-function at the head of each block *d* in *b*'s dominance frontier. The parallel execution semantics of the ϕ-functions lets the algorithm insert them at the head of *d* in any order. When it adds a ϕ-function for *x* to *d*, the algorithm adds *d* to *WorkList* to reflect the new assignment to *x* in *d*.

Example

Fig. 9.12 recaps our running example. Panel (a) shows the code and panel (b) shows the dominance frontiers for the CFG.



(a) Code for the Basic Blocks

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DF	∅	$\{B_1\}$	$\{B_3\}$	$\{B_1\}$	∅	$\{B_3\}$	$\{B_7\}$	$\{B_3\}$	$\{B_7\}$

(b) Dominance Frontiers in the CFG

Name	Blocks
a	$\{B_1, B_5\}$
b	$\{B_2, B_7\}$
c	$\{B_1, B_2, B_8\}$
d	$\{B_2, B_5, B_6\}$
i	$\{B_0, B_3\}$

(c) Blocks Sets for Each Global Name

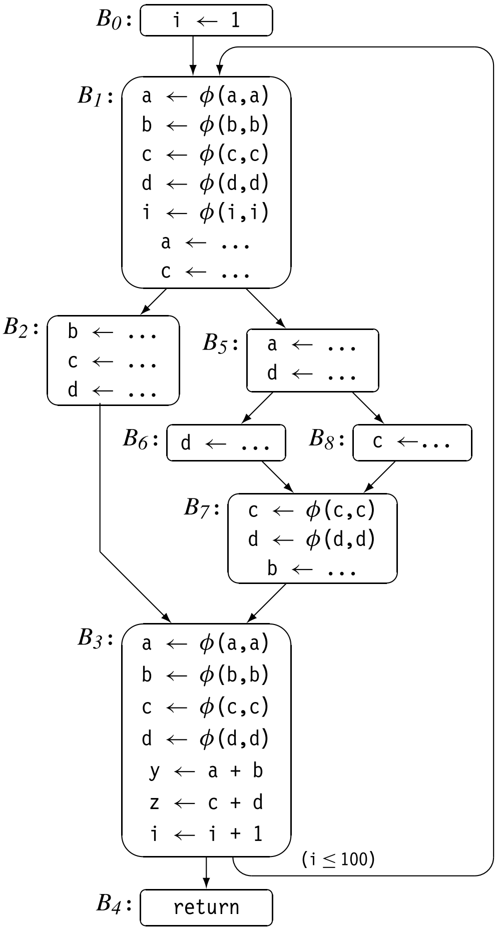
■ **FIGURE 9.12** Example Code for ϕ-Function Insertion.

The first step in the ϕ-function insertion algorithm finds global names and computes the *Blocks* set for each name. The global names are {a, b, c, d, i}. The *Blocks* sets for the global names are shown in panel (c). While the algorithm computes a *Blocks* set for each of y and z, the table omits them because they are not global names.

The compiler could avoid computing *Blocks* sets for nonglobal names, at the cost of another pass over the code.

The ϕ-function insertion algorithm, shown in Fig. 9.11(b), works on a name-by-name basis. Consider its actions for the variable a in the example. First, it initializes the worklist to *Blocks*(a) $= \{B_1, B_5\}$, to denote the fact that a is defined in B_1 and B_5.

The definition of a in B_1 causes insertion of a ϕ-function for a at the start of each block in DF(B_1) $= \{\, B_1 \,\}$. The ϕ-function in B_1 is a new assignment, so the algorithm adds B_1 to *WorkList*. Next, the algorithm removes B_5 from the worklist and inserts a ϕ-function in each block of DF(B_5) $= \{B_3\}$. The new ϕ-function in B_3 causes the algorithm to add B_3 to the worklist. When B_1 comes off the worklist, the algorithm discovers that the ϕ-function induced by B_1 in B_1 already exists. It neither adds a duplicate ϕ-function nor adds blocks to *Worklist*. When B_3 comes off the worklist, the algorithm also finds the ϕ-function for a in B_1. At that point, *WorkList* is empty and the processing for a halts.



```
B0: i ← 1

B1: a ← φ(a,a)
    b ← φ(b,b)
    c ← φ(c,c)
    d ← φ(d,d)
    i ← φ(i,i)
      a ← ...
      c ← ...

B2: b ← ...
    c ← ...
    d ← ...

B5: a ← ...
    d ← ...

B6: d ← ...

B8: c ←...

B7: c ← φ(c,c)
    d ← φ(d,d)
      b ← ...

B3: a ← φ(a,a)
    b ← φ(b,b)
    c ← φ(c,c)
    d ← φ(d,d)
      y ← a + b
      z ← c + d
      i ← i + 1

(i ≤ 100)

B4: return
```

■ **FIGURE 9.13** Example Code After ϕ-Function Insertion.

The algorithm follows the same logic for each name in *Globals*, to produce the following insertions:

	a	b	c	d	i
ϕ-functions	$\{B_1,B_3\}$	$\{B_1,B_3\}$	$\{B_1,B_3,B_7\}$	$\{B_1,B_3,B_7\}$	$\{B_1\}$

The resulting code appears in Fig. 9.13.

Limiting the algorithm to global names keeps it from inserting dead ϕ-functions for y and z in block B_1. ($B_1 \in \text{DF}(B_3)$ and B_3 defines both y and z.) However, the distinction between local names and global names is not sufficient to avoid all dead ϕ-functions. For example, the ϕ-function for b in B_1 is not live because b is redefined before its value is used. To avoid inserting these ϕ-functions, the compiler can construct LIVEOUT sets and add a test based on liveness to the inner loop of the ϕ-function insertion algorithm. That modification causes the algorithm to produce *pruned SSA form*.

Efficiency Improvements

To improve efficiency, the compiler should avoid two kinds of duplication. First, the algorithm should avoid placing any block on the worklist more than once per global name. It can keep a checklist of blocks that have already been processed for the current name and reset the checklist when it starts to process a new name.

Second, a given block can be in the dominance frontier of multiple nodes that appear on the *WorkList*. The algorithm must check, at each insertion, for a preexisting ϕ-function for the current name. Rather than searching through the ϕ-functions in the block, the compiler should maintain a checklist of blocks that already contain ϕ-functions for the current variable. Again, this checklist must be reset when the algorithm starts to process a new name.

Both of these checklists can be implemented as sparse sets (see Appendix B.2.3).

9.3.4 Renaming

Earlier, we stated that the algorithm for renaming variables was conceptually straightforward. The details, however, require explanation.

In the final SSA form, each global name becomes a base name, and individual definitions of that base name are distinguished by the addition of a numerical subscript. For a name that corresponds to a source-language variable, say a, the algorithm uses a as the base name. Thus, the first definition of a that the renaming algorithm encounters will be named a_0 and the second will be a_1. For a compiler-generated temporary, the algorithm can use its pre-SSA name as its base name.

The algorithm, shown in Fig. 9.14, renames both definitions and uses in a preorder walk over the procedure's dominator tree. In each block, it first renames the values defined by ϕ-functions at the head of the block. Next, it visits each operation in the block, in order. It rewrites the operands with current SSA names and then creates a new SSA name for the result of the operation. This latter act makes the new name current. After all the operations in the block have been rewritten, the algorithm rewrites the appropriate ϕ-function parameters in each CFG successor of the block, using the current SSA names. Finally, it recurs on any children of the block in the dominator tree. When it returns from those recursive calls, it restores the set of current SSA names to the state that existed before the current block was visited.

```
// Renaming algorithm

for each global name i do
    counter[i] ← 0
    stack[i] ← Ø

Rename( root of the CFG )


NewName(n)
    i ← counter[n]
    counter[n] ← counter[n] + 1
    push i onto stack[n]
    return "$n_i$"
```

```
Rename(b)
    for each $\phi$-function in b, "x ← $\phi(\cdots)$" do
        rewrite x as NewName(x)

    for each operation "x ← y op z" in b do
        if y ∈ Globals then
            rewrite y with subscript top(stack[y])
        if z ∈ Globals then
            rewrite z with subscript top(stack[z])
        if x ∈ Globals then
            rewrite x as NewName(x)

    for each successor of b in the CFG do
        fill in $\phi$-function parameters

    for each successor s of b in the dominator tree do
        Rename(s)

    for each operation "x ← y op z" in b and
        each $\phi$-function "x ← $\phi(\cdots)$" do
        pop(stack[x])
```

■ **FIGURE 9.14** Algorithm for Renaming After ϕ-Insertion.

To manage the names, the algorithm uses a counter and a stack for each global name. A name's stack holds the subscript from its current SSA name. At each definition, the algorithm generates a new subscript for the defined base name by pushing the value of its current counter onto the stack and incrementing the counter. Thus, the value on top of the stack for n is always the subscript of n's current SSA name.

As the final step, after recurring on the block's children in the dominator tree, the algorithm pops all the names generated in that block off their respective stacks. This action reveals the names that held at the end of that block's immediate dominator. Those names may be needed to process the block's remaining dominator-tree siblings.

The stack and the counter serve distinct and separate purposes. As the algorithm moves up and down the dominator tree, the stack is managed to simulate the lifetime of the most recent definition in the current block. The counter, on the other hand, grows monotonically to ensure that each successive definition receives a unique SSA name.

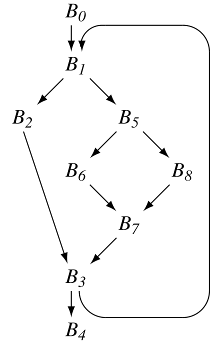
Fig. 9.14 summarizes the algorithm. It initializes the stacks and counters, then calls *Rename* on the dominator tree's root—the CFG's entry node. *Rename* processes the block, updates ϕ-function arguments in its CFG successor blocks, and recurs on its dominator-tree successors. To finish the block, *Rename* pops off the stacks any names that it added as it processed the block. The function *NewName* manipulates the counters and stacks to create new SSA names as needed.

One final detail remains. When *Rename* rewrites the ϕ-function parameters in each of b's CFG successors, it needs a mapping from b to an ordinal parameter slot in those ϕ-functions for b. That is, it must know which parameter slot in the ϕ-functions corresponds to b.

When we draw SSA form, we assume a left-to-right order that matches the left-to-right order in which the edges are drawn. Internally, the compiler can number the edges and parameter slots in any consistent fashion that produces the desired result. This requires cooperation between the code that builds SSA and the code that builds the CFG. (For example, if the CFG implementation uses a list of edges leaving each block, the order of that list can determine the mapping.)

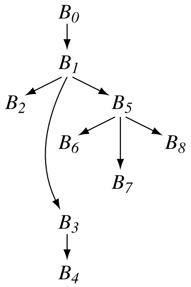
Example

To finish the continuing example, let's apply the renaming algorithm to the code in Fig. 9.13. Assume that a_0, b_0, c_0, and d_0 are defined on entry to B_0. Fig. 9.15 shows the states of the counters and stacks for global names at various points during the process.



Example CFG

The algorithm makes a preorder walk over the dominator tree, which, in this example, corresponds to visiting the nodes in ascending order by name. Fig. 9.15(a) shows the initial state of the stacks and counters. As the algorithm proceeds, it takes the following actions:

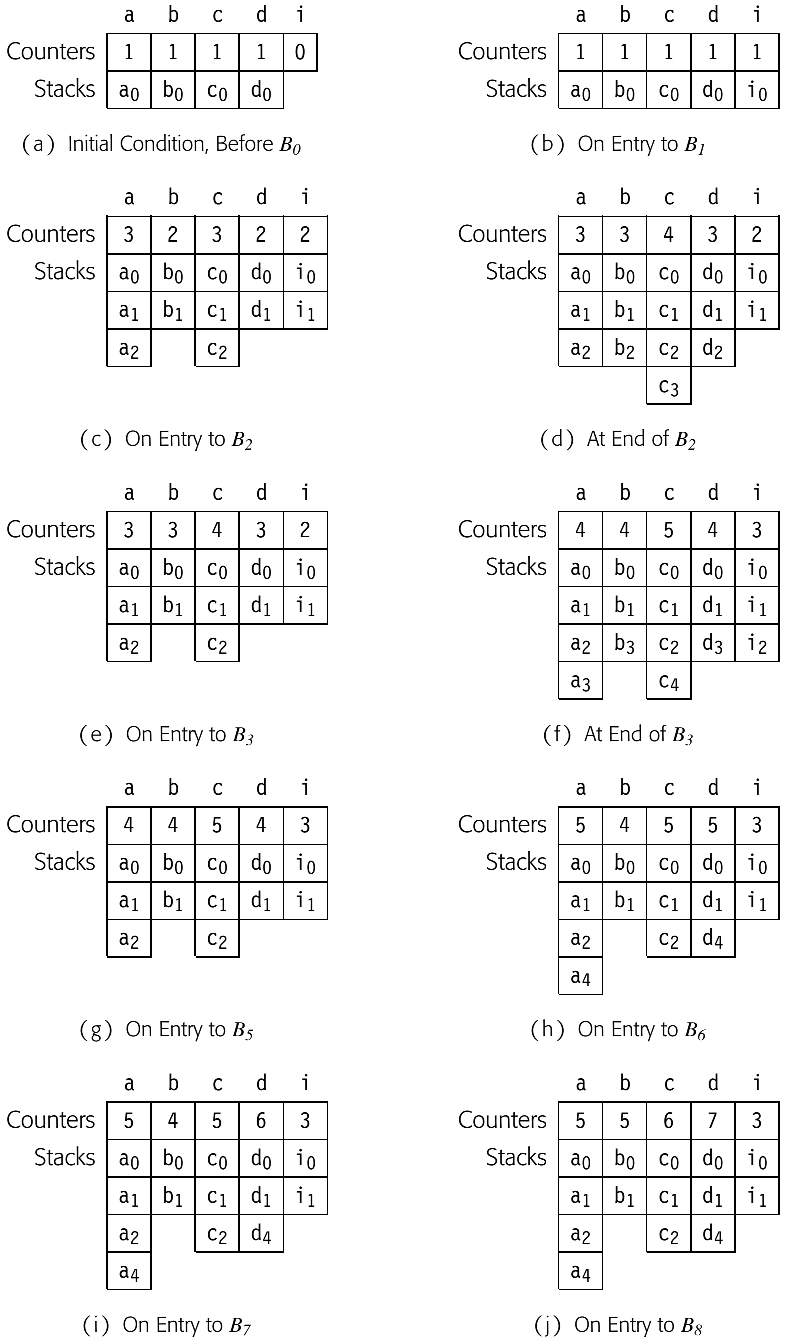


Example Dominator Tree

Block B_0 This block contains only one operation. *Rename* rewrites i with i_0, increments i's counter, and pushes i_0 onto the stack for i. Next, it visits B_0's CFG-successor, B_1, and rewrites the ϕ-function parameters that correspond to B_0 with their current names: a_0, b_0, c_0, d_0, and i_0. It then recurs on B_0's child in the dominator tree, B_1. After that, it pops the stack for i and returns.

Block B_1 *Rename* enters B_1 with the state shown in panel (b). It rewrites the ϕ-function targets with new names, a_1, b_1, c_1, d_1, and i_1. Next, it creates new names for the definitions of a and c and rewrites them. Neither of B_1's CFG successors have ϕ-functions, so it recurs on B_1's dominator-tree children, B_2, B_3, and B_5. Finally, it pops the stacks and returns.

Block B_2 *Rename* enters B_2 with the state shown in panel (c). This block has no ϕ-functions to rewrite. *Rename* rewrites the definitions of b, c, and d, creating a new SSA name for each. It then rewrites ϕ-function parameters in B_2's CFG successor, B_3. Panel (d) shows the stacks and counters just before they are popped. Finally, it pops the stacks and returns.



	a	b	c	d	i
Counters	1	1	1	1	0
Stacks	a_0	b_0	c_0	d_0	

(a) Initial Condition, Before B_0

	a	b	c	d	i
Counters	1	1	1	1	1
Stacks	a_0	b_0	c_0	d_0	i_0

(b) On Entry to B_1

	a	b	c	d	i
Counters	3	2	3	2	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

(c) On Entry to B_2

	a	b	c	d	i
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_2	c_2	d_2	
			c_3		

(d) At End of B_2

	a	b	c	d	i
Counters	3	3	4	3	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

(e) On Entry to B_3

	a	b	c	d	i
Counters	4	4	5	4	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2	b_3	c_2	d_3	i_2
	a_3		c_4		

(f) At End of B_3

	a	b	c	d	i
Counters	4	4	5	4	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

(g) On Entry to B_5

	a	b	c	d	i
Counters	5	4	5	5	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

(h) On Entry to B_6

	a	b	c	d	i
Counters	5	4	5	6	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

(i) On Entry to B_7

	a	b	c	d	i
Counters	5	5	6	7	3
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2	d_4	
	a_4				

(j) On Entry to B_8

■ **FIGURE 9.15** States in the Renaming Example.

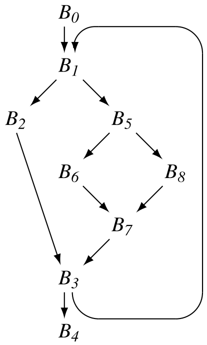
Block B_3 *Rename* enters B_3 with the state shown in panel (e). Notice that the stacks have been popped to their state when *Rename* entered B_2, but the counters reflect the names created inside B_2. In B_3, *Rename* rewrites the ϕ-function targets, creating new SSA names for each. Next, it rewrites each assignment in the block, using current SSA names for the uses of global names and then creating new SSA names for definitions of global names.

Since y and z are not global names, the renamer does not change them.

B_3 has two CFG successors, B_1 and B_4. In B_1, it rewrites the ϕ-function parameters that correspond to the edge from B_3, using the stacks and counters shown in panel (f). B_4 has no ϕ-functions. Next, *Rename* recurs on B_3's dominator-tree child, B_4. When that call returns, *Rename* pops the stacks and returns.

Block B_4 This block just contains a return statement. It has no ϕ-functions, definitions, uses, or successors in either the CFG or the dominator tree. Thus, *Rename* performs no actions and leaves the stacks and counters unchanged.

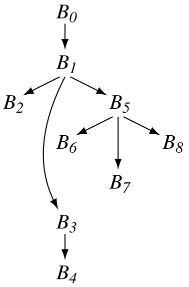
Block B_5 After B_4, *Rename* pops through B_3 back to B_1. With the stacks as shown in panel (g), it recurs down into B_1's final dominator-tree child, B_5. B_5 has no ϕ-functions. *Rename* rewrites the two assignment statements, creating new SSA names as needed. Neither of B_5's CFG successors has ϕ-functions. *Rename* next recurs on B_5's dominator-tree children, B_6, B_7, and B_8. Finally, it pops the stacks and returns.



Example CFG

Block B_6 *Rename* enters B_6 with the state in panel (h). B_6 has no ϕ-functions. *Rename* rewrites the assignment to d, generating the new SSA name d_5. Next, it visits the ϕ-functions in B_6's CFG successor B_7. It rewrites the ϕ-function arguments along the edge from B_6 with their current names, c_2 and d_5. Since B_6 has no dominator-tree children, it pops the stack for d and returns.

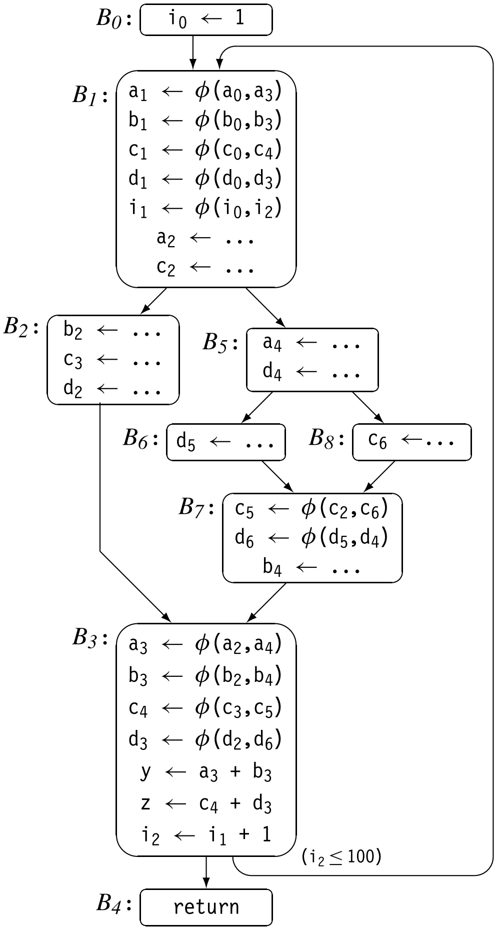
Block B_7 *Rename* enters B_7 with the state shown in panel (i). It first renames the ϕ-function targets with new SSA names, c_5 and d_6. Next, it rewrites the assignment to b with new SSA name b_4. It then rewrites the ϕ-function arguments in B_7's CFG successor, B_3, with their current names. Since B_7 has no dominator-tree children, it pops the stacks and returns.



Example Dominator Tree

Block B_8 *Rename* enters B_8 with the state shown in panel (j). B_8 has no ϕ-functions. *Rename* rewrites the assignment to c with new SSA name c_6. It rewrites the appropriate ϕ-function arguments in B_7 with their current names, c_6 and d_4. Since B_8 has no dominator-tree children, it pops the stacks and returns.

Fig. 9.16 shows the code after *Rename* halts.


B_0: $i_0 \leftarrow 1$
B_1: $a_1 \leftarrow \phi(a_0,a_3)$
$b_1 \leftarrow \phi(b_0,b_3)$
$c_1 \leftarrow \phi(c_0,c_4)$
$d_1 \leftarrow \phi(d_0,d_3)$
$i_1 \leftarrow \phi(i_0,i_2)$
$a_2 \leftarrow ...$
$c_2 \leftarrow ...$
B_2: $b_2 \leftarrow ...$
$c_3 \leftarrow ...$
$d_2 \leftarrow ...$
B_5: $a_4 \leftarrow ...$
$d_4 \leftarrow ...$
B_6: $d_5 \leftarrow ...$
B_8: $c_6 \leftarrow ...$
B_7: $c_5 \leftarrow \phi(c_2,c_6)$
$d_6 \leftarrow \phi(d_5,d_4)$
$b_4 \leftarrow ...$
B_3: $a_3 \leftarrow \phi(a_2,a_4)$
$b_3 \leftarrow \phi(b_2,b_4)$
$c_4 \leftarrow \phi(c_3,c_5)$
$d_3 \leftarrow \phi(d_2,d_6)$
$y \leftarrow a_3 + b_3$
$z \leftarrow c_4 + d_3$
$i_2 \leftarrow i_1 + 1$
$(i_2 \leq 100)$
B_4: return

■ **FIGURE 9.16** Example Code After Renaming.

A Final Improvement

We can reduce the time and space spent in stack manipulation with a clever implementation of *NewName*. The primary use of the stacks is to reset the name space on exit from a block. If a block redefines the same base name multiple times, the stack only needs to keep the most recent name. For example, in block B_1, both a and c are defined twice. *NewName* could reuse the slots for a_1 and c_1 when it creates a_2 and c_2.

With this change, *Rename* performs one push and one pop per base name defined in the block. *NewName* can keep a list of the stack entries that it creates; on exit from the block, *Rename* can then walk the list to pop the appropriate stacks. The stacks require less space; their size is bounded by the depth of the dominator tree. Stack manipulation is simplified; the algorithm performs fewer push and pop operations and the push operation need not test for a stack overflow.

9.3.5 Translation out of SSA Form

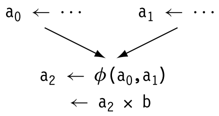
Actual processors do not implement ϕ-functions, so the compiler must rewrite the code without the ϕ-functions.

A compiler that uses SSA form must translate that form of the code back into a more conventional model—one without ϕ-functions—before the code can execute on conventional computer hardware. The compiler must replace the ϕ-functions with copy operations and place them in the code so that they reproduce the semantics of those ϕ-functions: both the control-based selection of values and the parallel execution at the start of the block.

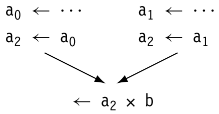
This section addresses out-of-SSA translation. It begins with an overly simple, or naive, translation, which informs and motivates the actual translation schemes. Next, it presents two example problems that demonstrate the problems that can arise in translating from SSA form back to conventional code. Finally, it presents a unified framework that addresses the known complexities of the translation.

The Naive Translation

A ϕ-function is just a copy operation that selects its input based on prior control-flow. To replicate the effect of a ϕ-function at the top of block b, the compiler can insert, at the end of each CFG-predecessor of b, a copy operation that moves the appropriate ϕ-function argument into the name defined by the ϕ-function (shown in the margin). Once the compiler has inserted the copies, it can delete the ϕ-function.



Code in SSA Form

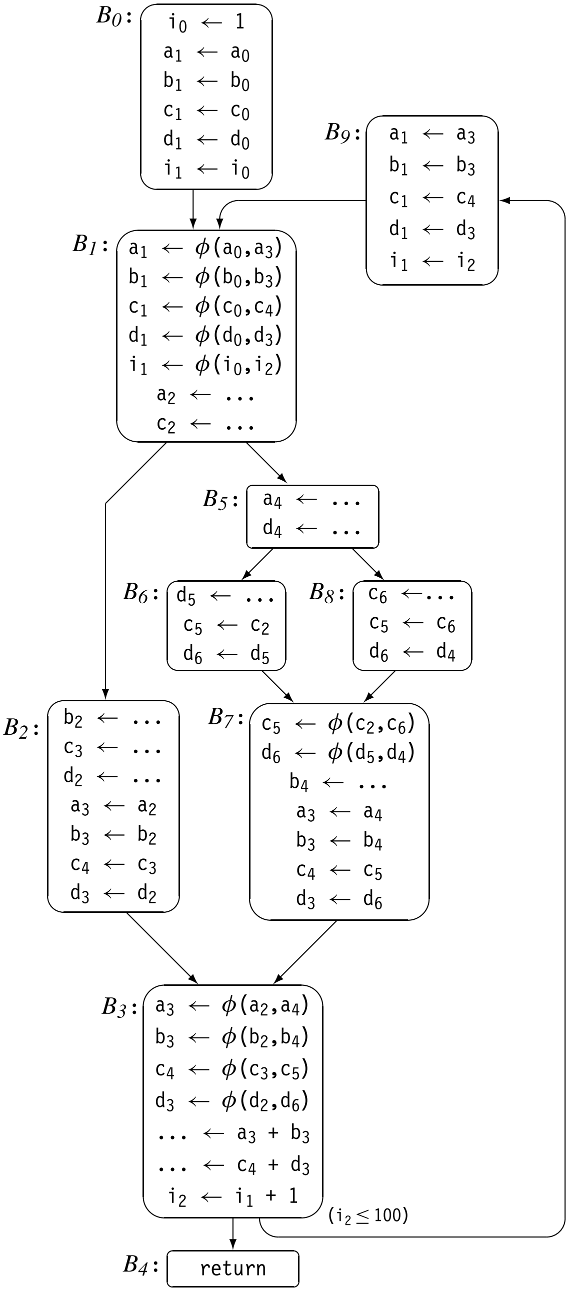


Code After Translation
Out of SSA Form

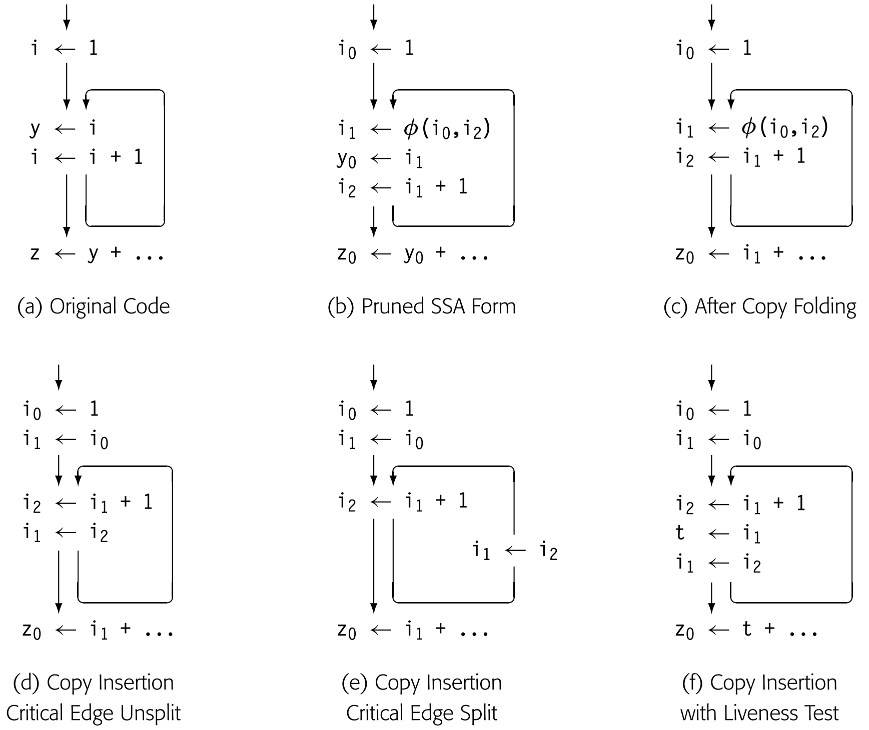
This process, while conceptually simple, has some complications. Consider, for example, the continuing example from Fig. 9.16. Three blocks in the CFG contain ϕ-functions: B_1, B_3, and B_7. Fig. 9.17 shows the code after copies have been inserted.

For B_3 and B_7, insertion into the predecessor blocks works. The predecessors of both B_3 and B_7 have one successor each, so the copy operations inserted at the end of those predecessor blocks have no effect on any path other than the one to the ϕ-function.

The situation is more complex for B_1. Copy insertion at the end of B_1's predecessor, B_0, produces the desired result; the copies only occur on the path (B_0, B_1). With B_1's other predecessor, B_3, simple insertion will not work. A copy inserted at the end of B_3 will execute on both (B_3, B_1) and (B_3, B_4). Along (B_3, B_4), the copy operation may change a value that is live in B_4.



■ **FIGURE 9.17** Example After Copy Insertion.



(a) Original Code

(b) Pruned SSA Form

(c) After Copy Folding

(d) Copy Insertion Critical Edge Unsplit

(e) Copy Insertion Critical Edge Split

(f) Copy Insertion with Liveness Test

■ **FIGURE 9.18** An Example of the Lost-Copy Problem.

The edge (B_3, B_1) highlights a more general problem with code placement on a *critical edge*. B_3 has multiple successors, so the compiler cannot insert the copy at the end of B_3. B_1 has multiple predecessors, so the compiler cannot insert the copy at the start of B_1. Since neither solution works, the compiler must split the edge and create a new block to hold the inserted copy operations. With the split edge and the creation of B_9, the translated code faithfully reproduces the effects of the SSA form of the code.

Critical edge
A flow graph edge (*i*,*j*) is a critical edge if *i* has multiple successors and *j* has multiple predecessors.

Optimizations that move or insert code often need to split critical edges.

Problems with the Naive Translation

If the compiler applies the naive translation to code that was produced directly by the translation into SSA form, the results will be correct, as long as critical edges can be split. If, however, the compiler transforms the code while it is in SSA form—particularly, transformations that move definitions or uses of SSA names—or if the compiler cannot split critical edges, then the naive translation can produce incorrect code. Two examples demonstrate how the naive translation can fail.

The Lost-Copy Problem

In Fig. 9.17, the compiler had to split the edge (B_3, B_1) to create a location for the copy operations associated with that edge. In some situations, the compiler cannot or should not split a critical edge. For example, an SSA-based register allocator should not add any blocks or edges during copy insertion (see Section 13.5.3). The combination of an unsplit critical edge and an optimization that extends some SSA-name's live range can create a situation where naive copy insertion fails.

Fig. 9.18(a) shows an example to demonstrate the problem. The loop increments i. The computation of z after the loop uses the second-to-last value of i. Panel (b) shows the pruned SSA for the code.

Copy folding
an optimization that removes unneeded copy operations by renaming the source and destination to the same name, when such renaming does not change the flow of values

Copy folding is also called *copy coalescing* (see Section 13.4.3).

Panel (c) shows the code after copy folding. The use of y_0 in the computation of z_0 has been replaced with a use of i_1. The last use of i_1 in panel (b) was in the assignment to y_0; folding the copy extends the live range of i_1 beyond the end of the loop in panel (c).

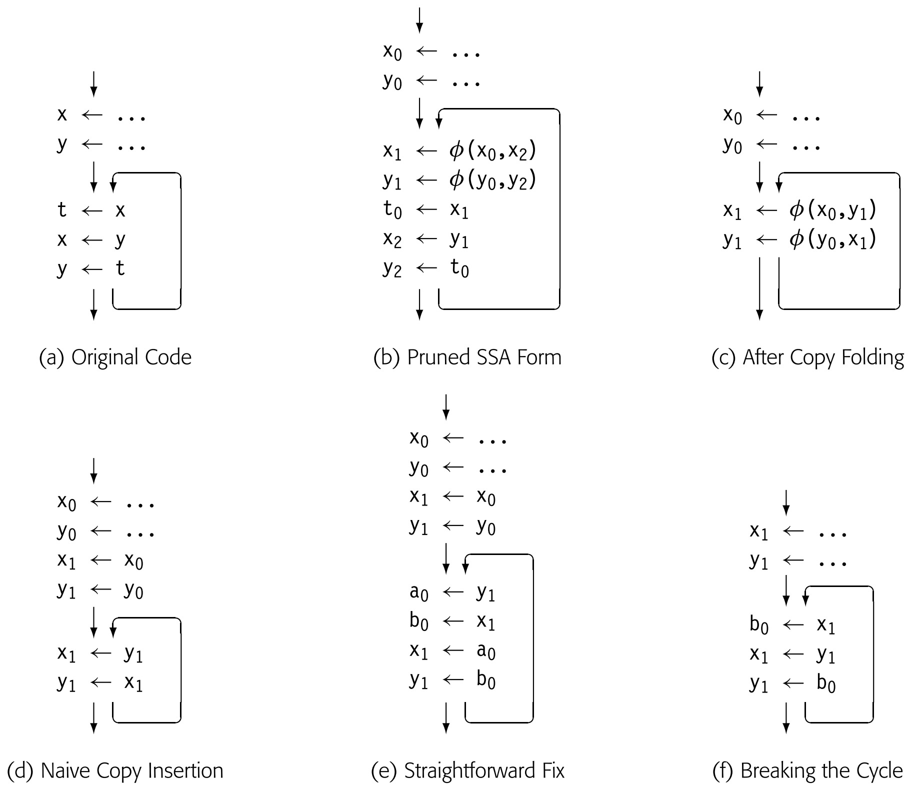
Copy insertion on the code in panel (c) adds $i_1 \leftarrow i_0$ to the end of the preloop block, and $i_1 \leftarrow i_2$ at the end of the loop. Unfortunately, that latter assignment kills the value in i_1; the computation of z_0 now receives the final value of i rather than its penultimate value. Copy insertion produces incorrect code because it extends i_1's live range.

Splitting the critical edge cures the problem, as shown in panel (e); the copy does not execute on the loop's final iteration. When the compiler cannot split that edge, it must add a new name to preserve the value of i_1, as shown in panel (f). A simple, ad-hoc addition to the copy insertion process can avoid the lost-copy problem. As the compiler inserts copies, it should check whether or not the target of the new copy is live at the insertion point. If the target is live, the compiler must introduce a new name, copy the live value into it, and propagate that name to the uses after the insertion point.

The Swap Problem

The concurrent semantics of ϕ-functions create another problem for out-of-SSA translation, which we call the swap problem. The motivating example appears in Fig. 9.19(a): a simple loop that repeatedly swaps the values of x and y. If the compiler builds pruned SSA-form, as in panel (b), and performs copy folding, as in panel (c), it creates a valid program in SSA form that relies directly on the concurrent semantics of the ϕ-functions in a single block.

Because the two ϕ-functions read their values concurrently and then write their results concurrently, the code in panel (c) has the same meaning as the



(a) Original Code

(b) Pruned SSA Form

(c) After Copy Folding

(d) Naive Copy Insertion

(e) Straightforward Fix

(f) Breaking the Cycle

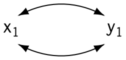
■ **FIGURE 9.19** An Example of the Swap Problem.

original code from panel (a). Naive copy-insertion, however, replaces each ϕ-function with a sequential copy operation, as shown in panel (d). The two sequential copies have a different result than did the two ϕ-functions; the substitution fundamentally changes the meaning of the code.

To maintain the original code's meaning, the compiler must ensure that the inserted copies faithfully reproduce the flow of values specified by the ϕ-functions. Thus, it must pay attention to any values that are defined by one ϕ-function and used by another ϕ-function in the same block.

In some cases, the compiler must introduce one or more new names. The straightforward solution to this problem is to adopt a two-stage copy protocol, as shown in panel (e). The first stage copies each of the ϕ-function arguments into its own temporary name, simulating the control-based selection and the parallel read of the ϕ-function. The second stage then copies those values to the ϕ-function targets.

Unfortunately, this solution doubles the number of copy operations required to translate out of SSA form. The compiler can reduce the number of temporary names and extra copy operations by building a small dependence graph for the set of parallel copies implied by the ϕ-functions and using the graph to guide insertion of the sequential copies. If the dependence graph is acyclic, then the compiler can use it to schedule the copy operations in a way that requires no additional names or operations (see Chapter 12).



If the dependence graph contains cycles, then the compiler must break each cycle with a copy into a name not involved in the cycle. This may require a new name. The dependence graph for the example, shown in the margin, consists of a two node cycle. It requires one new name to break the cycle, which produces the code shown in panel (f).

A Unified Approach to Out-of-SSA Translation

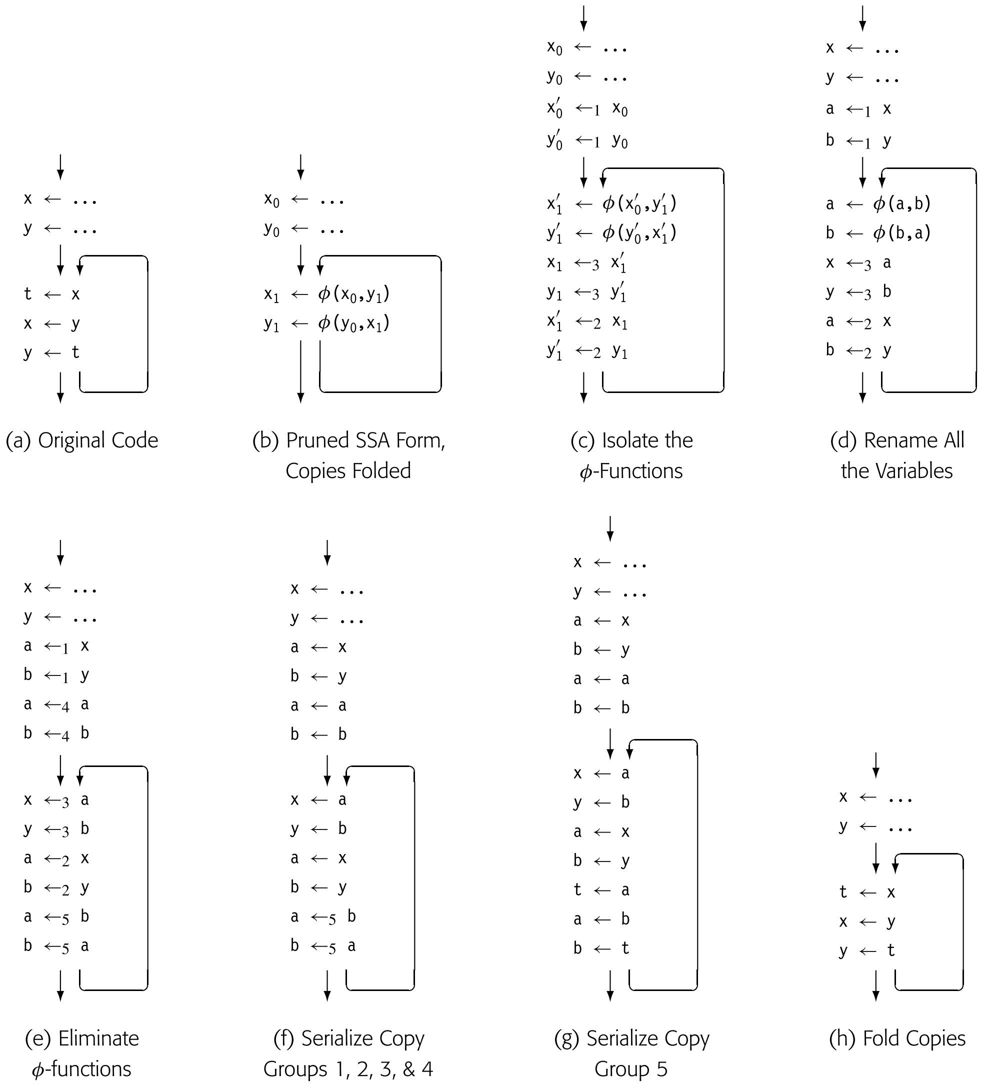
The swap problem and the copy problem arise from two distinct phenomena: transformations that change the range over which an SSA-name is live, and failure to preserve the parallel semantics of ϕ-function execution during translation out of SSA-form. Common code transformations, such as copy folding, code motion, and cross-block instruction scheduling, can create the circumstances that trigger these problems. While the solutions proposed in the previous section will generate correct code, neither solution provides a clean framework for understanding the underlying issues.

The unified approach uses a three-phase plan to address the two issues caused by code transformations on the SSA form: changes in the live ranges of SSA names and implicit use of the parallel semantics of ϕ-function execution. Phase one introduces a new set of names to isolate ϕ-functions from the rest of the code; it then inserts parallel copy operations to connect those names with the surrounding context. Phase two replaces ϕ-functions with parallel copy operations in predecessor blocks. Phase three rewrites each block of parallel copies with an equivalent series of sequential copies. This process avoids both the swap problem and the lost copy problem. At the same time, it eliminates the need to split critical edges.

Phase One

To isolate the name space for a ϕ-function, such as $a_0 \leftarrow \phi(a_1, a_2, \ldots, a_n)$, phase one rewrites it as $a'_0 \leftarrow \phi(a'_1, a'_2, \ldots, a'_n)$. To connect the new primed names with the surrounding code, the compiler adds a copy operation $a'_i \leftarrow a_i$ to the end of the predecessor block associated with a_i, for each parameter a_i. To retain the parallel execution semantics of the ϕ-functions, the compiler will use parallel copy groups for the copies that it inserts.

We will denote a parallel copy group by adding a common subscript to the assignment operator, $\leftarrow_i$.



$x \leftarrow \ldots$
$y \leftarrow \ldots$

$t \leftarrow x$
$x \leftarrow y$
$y \leftarrow t$

(a) Original Code

$x_0 \leftarrow \ldots$
$y_0 \leftarrow \ldots$

$x_1 \leftarrow \phi(x_0,y_1)$
$y_1 \leftarrow \phi(y_0,x_1)$

(b) Pruned SSA Form, Copies Folded

$x_0 \leftarrow \ldots$
$y_0 \leftarrow \ldots$
$x'_0 \leftarrow_1 x_0$
$y'_0 \leftarrow_1 y_0$

$x'_1 \leftarrow \phi(x'_0,y'_1)$
$y'_1 \leftarrow \phi(y'_0,x'_1)$
$x_1 \leftarrow_3 x'_1$
$y_1 \leftarrow_3 y'_1$
$x'_1 \leftarrow_2 x_1$
$y'_1 \leftarrow_2 y_1$

(c) Isolate the ϕ-Functions

$x \leftarrow \ldots$
$y \leftarrow \ldots$
$a \leftarrow_1 x$
$b \leftarrow_1 y$

$a \leftarrow \phi(a,b)$
$b \leftarrow \phi(b,a)$
$x \leftarrow_3 a$
$y \leftarrow_3 b$
$a \leftarrow_2 x$
$b \leftarrow_2 y$

(d) Rename All the Variables

$x \leftarrow \ldots$
$y \leftarrow \ldots$
$a \leftarrow_1 x$
$b \leftarrow_1 y$
$a \leftarrow_4 a$
$b \leftarrow_4 b$

$x \leftarrow_3 a$
$y \leftarrow_3 b$
$a \leftarrow_2 x$
$b \leftarrow_2 y$
$a \leftarrow_5 b$
$b \leftarrow_5 a$

(e) Eliminate ϕ-functions

$x \leftarrow \ldots$
$y \leftarrow \ldots$
$a \leftarrow x$
$b \leftarrow y$
$a \leftarrow a$
$b \leftarrow b$

$x \leftarrow a$
$y \leftarrow b$
$a \leftarrow x$
$b \leftarrow y$
$a \leftarrow_5 b$
$b \leftarrow_5 a$

(f) Serialize Copy Groups 1, 2, 3, & 4

$x \leftarrow \ldots$
$y \leftarrow \ldots$
$a \leftarrow x$
$b \leftarrow y$
$a \leftarrow a$
$b \leftarrow b$

$x \leftarrow a$
$y \leftarrow b$
$a \leftarrow x$
$b \leftarrow y$
$t \leftarrow a$
$a \leftarrow b$
$b \leftarrow t$

(g) Serialize Copy Group 5

$x \leftarrow \ldots$
$y \leftarrow \ldots$

$t \leftarrow x$
$x \leftarrow y$
$y \leftarrow t$

(h) Fold Copies

■ **FIGURE 9.20** Unified Approach to Out of SSA Translation.

After the group of ϕ-functions at the head of a block, the compiler should insert another parallel copy group. For each ϕ-function in the block, $a_i \leftarrow \phi(\ldots)$, the copy group should include a copy of the form $a_i \leftarrow a'_i$. The net effect of these three actions is to isolate the names used in the ϕ-functions from the surrounding code and to make the impact of parallel execution explicit, outside of the ϕ-functions.

Fig. 9.20 shows the effects of this transformation on the example from the swap problem. Panel (a) shows the original code; panel (b) shows it in pruned SSA form, with copies folded. Panel (c) shows the code after the compiler has isolated the ϕ-functions. The ϕ-function parameters have been renamed and parallel copy groups inserted.

- Parallel copy group 1, at the end of the first block, gives x'_0 and y'_0 their initial values.
- Parallel copy group 2, at the end of the loop body, gives x'_1 and y'_1 their values from computation inside the loop. (The loop body is its own predecessor.)
- Parallel copy group 3, after the ϕ-functions, copies the values defined by the ϕ-functions into the names that they had before the renaming transformation.

At this point, the compiler can rename all of the primed variables and drop all of the subscripts from SSA names, as shown in panel (d). The renamed code retains the meaning of the original code.

Phase Two

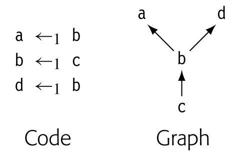
This phase replaces ϕ-functions by inserting copies into predecessor blocks and deleting the ϕ-functions. To retain the ϕ-function semantics, the compiler uses parallel copies in each block.

At the end of phase one as shown in panel (d), the actual value swap occurs during evaluation of the ϕ-function arguments. After ϕ-function replacement, shown in panel (e), that value swap occurs in parallel copy group 5, at the end of the loop body.

At the end of phase two, the compiler has eliminated all of the ϕ-functions. The code still contains groups of parallel copy operations that implement the semantics of the ϕ-functions. To complete the process, the compiler must rewrite each parallel copy group into a set of serial copies. The code will likely contain multiple (perhaps many) unneeded copy operations. Coalescing can eliminate some or all of them (see Section 13.4.3).

Phase Three

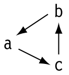
The final phase examines each parallel copy group and rewrites it with an equivalent group of sequential copy operations. It builds a data-dependence graph for the copy group (see Section 4.3.2). If the graph is acyclic, as in the acyclic graph shown in the margin, the compiler can simply insert copies in the order implied by the graph—leaves to roots. For the first example, the graph requires that a ← b and d ← b precede b ← c.



$a \leftarrow_1 b$
$b \leftarrow_1 c$
$d \leftarrow_1 b$

Acyclic Dependence Graph

$\texttt{a} \leftarrow_2 \texttt{b}$
$\texttt{b} \leftarrow_2 \texttt{c}$
$\texttt{c} \leftarrow_2 \texttt{a}$



Code Graph

Cyclic Dependence Graph

If the dependence graph contains a cycle, as shown in the example in the margin, the compiler must insert copies in a way that breaks the cycle. In the example, it must copy one of the values, say a, into a new temporary name, say t. Then, it can perform a ← b and b ← c. It can finish the copy group with c ← t. This breaks the cycle and correctly implements the semantics of teh parallel copy group.

In some cases, the compiler can avoid the new name by careful ordering. For example, if the second example also included a copy d $\leftarrow_2$ a, the compiler could schedule d ← a first and break the cycle by rewriting c ← a as c ← d.

In the example, groups 1, 2, 3, and 4 can be serialized without additional names, as shown in panel (f). Copy group 5 contains a cycle, so it requires one new name, t. Panel (g) shows the rewrite of copy group 5. Panel (h) shows the final code after copy folding.

9.3.6 **Using SSA Form**

A compiler writer uses SSA form because it improves the quality of analysis, the quality of optimization, or both. To see how analysis on SSA differs from the classical data-flow analysis techniques presented in Section 9.2, consider the problem of global constant propagation on SSA, using an algorithm called *sparse simple constant propagation* (SSCP).

In SSCP, the compiler annotates each SSA name with a value. The set of possible values forms a *semilattice*. A semilattice consists of a set L of values and a meet operator, $\wedge$. The meet operator must be idempotent, commutative, and associative; it imposes an order on the elements of L:

$$a \geq b \quad \text{if and only if} \quad a \wedge b = b, \text{ and}$$
$$a > b \quad \text{if and only if} \quad a \geq b \text{ and } a \neq b$$

Every semilattice has a bottom element, $\bot$, with the properties that

$$\forall\, a \in L, a \wedge \bot \;=\; \bot, \quad \text{and} \quad \forall\, a \in L, a \geq \bot.$$

Some semilattices also have a top element, $\top$, with the properties that

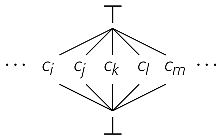
$$\forall\, a \in L, a \wedge \top \;=\; a \quad \text{and} \quad \forall\, a \in L, \top \geq a.$$

Semilattice
a set L and a *meet* operator $\wedge$ such that, $\forall$ a, b, and $c \in L$,

1. $a \wedge a \;=\; a,$
2. $a \wedge b \;=\; b \wedge a$, and
3. $a \wedge (b \wedge c) \;=\; (a \wedge b) \wedge c$

Compilers use semilattices to model the data domains of analysis problems.

In constant propagation, the structure of the semilattice used to model program values plays a critical role in the algorithm's runtime complexity. The semilattice for a single SSA name appears in the margin. It consists of $\top$, $\bot$, and an infinite set of distinct constant values. For any value x: $x \wedge \top = x$, and $x \wedge \bot = \bot$. For two constants, c_i and c_j: $c_i \wedge c_j = \bot$ if $c_i \neq c_j$. If $c_i = c_j$, then $c_i \wedge c_j = c_i$.



Semilattice for Constant Propagation

```
// Initialization Phase
WorkList ← $\emptyset$

for each SSA name n do
    initialize Value(n) by rules specified in the text
    if Value(n) $\neq \top$ then
        WorkList ← WorkList $\cup$ {n}

// Propagation Phase - Iterate to a fixed point
while (WorkList $\neq \emptyset$) do
    remove some n from WorkList  // Pick an arbitrary name
    for each operation op that uses n do
        let m be the SSA name that op defines
        if Value(m) $\neq \bot$ then          // Recompute and test for change
            t ← Value(m)
            Value(m) ← result of interpreting op over lattice values
            if Value(m) $\neq$ t then
                WorkList ← WorkList $\cup$ {m}
```

■ **FIGURE 9.21** Sparse Simple Constant Propagation Algorithm.

The algorithm for SSCP, shown in Fig. 9.21, consists of an initialization phase and a propagation phase. The initialization phase iterates over the SSA names. For each SSA name n, it examines the operation that defines n and sets *Value*(n) according to a simple set of rules.

1. If n is defined by a ϕ-function, SSCP sets *Value*(n) to $\top$.
2. if n's value is not known, SSCP sets *Value*(n) to $\top$.
3. If n's value is a known constant c_i, SSCP sets *Value*(n) to c_i.
4. If n's value *cannot be known*—for example, it is defined by reading a value from external media—SSCP sets *Value*(n) to $\bot$.

If *Value*(n) is not $\top$, the algorithm adds n to the worklist.

These initializations highlight the use of $\top$ and $\bot$ in the constant propagation semilattice. $\top$ indicates that the compiler does not yet know anything about the value, but that it might discover information about its value in the future. By contrast, $\bot$ indicates that the compiler has proven that the value is not a constant. For any SSA name m, *Value*(m) can change at most twice—the height of the semilattice. If *Value*(m) starts as $\top$, it can progress to some constant c_i or to $\bot$. If *Value*(m) is some constant c_i, it can progress to $\bot$. Once *Value*(m) is $\bot$, it cannot change.

The propagation phase is straightforward. It removes an SSA name *n* from the worklist. The algorithm examines each operation *op* that uses *n*, where *op* defines some SSA name *m*. If *Value(m)* has already reached $\bot$, then no further evaluation is needed. Otherwise, it models the evaluation of *op* by interpreting the operation over the lattice values of its operands. If the result is lower in the lattice than *Value(m)*, it lowers *Value(m)* accordingly and adds *m* to the worklist. The algorithm halts when the worklist is empty.

Interpreting an operation over lattice values requires some care. For a ϕ-function, the result is simply the meet of the lattice values of all the ϕ-function's arguments; the rules for meet are shown in the margin, in order of precedence. For other kinds of operations, the compiler needs a set of rules. Consider, for example, $a \times b$. If $a = 4$ and $b = 17$, then $a \times b = 68$. However, if $a = \bot$, then $a \times b = \bot$, unless $b = 0$. ($a \times 0 = 0$, for any a.)

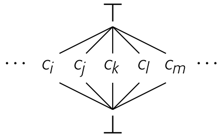
$$\begin{array}{ll} \top \wedge x = x & \forall\, x \\ \bot \wedge x = \bot & \forall\, x \\ c_i \wedge c_j = c_i & \text{if } c_i = c_j \\ c_i \wedge c_j = \bot & \text{if } c_i \neq c_j \end{array}$$

Rules for Meet

In general, the evaluation rules for operators should preserve $\top$, unless the other operand forces a value, as with multiplication by zero. If $a = \top$, then evaluating $a + b$ to $\top$ will defer determination of the sum until a's value is resolved to either a constant or $\bot$.

Complexity

The propagation phase of SSCP is a classic fixed-point scheme. The arguments for termination and complexity follow from the length of descending chains through the semilattice, shown again in the margin. The lattice value for an SSA name can change at most twice: from $\top$ to c_i and from c_i to $\bot$.



Semilattice for Constant Propagation

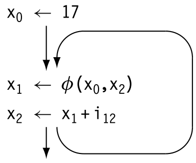
SSCP only adds an SSA name to the worklist when its value changes, so each name appears on the worklist at most twice. SSCP evaluates an operation when one of its operands is removed from the worklist, which bounds the number of evaluations at twice the number of uses in the code.

Optimism: The Role of Top

As discussed earlier, SSCP uses the lattice value $\top$ to represent a lack of knowledge. This practice differs from the data-flow problems in Section 9.2, which use the value $\bot$ but not the value $\top$. The use of $\top$ as an initial value plays a critical role in constant propagation; it allows values to propagate into cycles in the graph.

Because it initializes unknown values to $\top$, rather than $\bot$, it can propagate some values into cycles in the graph—loops in the CFG. Algorithms that begin with the value $\top$, rather than $\bot$, are often called *optimistic* algorithms. The intuition behind "optimism" is that initialization to $\top$ allows the algorithm to propagate information into a cyclic region, optimistically assuming that the value along the back edge will confirm this initial propagation. An initialization to $\bot$, called *pessimistic*, disallows that possibility.

Consider the SSA fragment in Fig. 9.22. If the algorithm initializes x_1 and x_2 to $\bot$ (pessimism), it will not propagate the value 17 into the loop. When it



```
x0 ← 17

x1 ← φ(x0,x2)
x2 ← x1 + i12
```

(a) The Code Fragment

	Lattice Values					
Time	Pessimistic			Optimistic		
Step	x_0	x_1	x_2	x_0	x_1	x_2
0	17	$\bot$	$\bot$	17	$\top$	$\top$
1	17	$\bot$	$\bot$	17	17	$17 + \texttt{i}_{12}$

(b) Results of Pessimistic and Optimistic Analyses

■ **FIGURE 9.22** Optimistic Constant Example.

evaluates the ϕ-function, it sets $\texttt{x}_1$ to $17 \wedge \bot = \bot$. Once $\texttt{x}_1 = \bot$, propagation sets $\texttt{x}_2 = \bot$, independent of $\texttt{i}_{12}$'s value.

If, on the other hand, the algorithm initializes $\texttt{x}_1 = \top$ and $\texttt{x}_2 = \top$ (optimism), it can propagate $\texttt{x}_0$'s value into the loop. It computes $\texttt{x}_1$'s value as $17 \wedge \top = 17$. Since $\texttt{x}_1$'s value has changed, the algorithm places $\texttt{x}_1$ on *WorkList*. The algorithm then reevaluates the definition of $\texttt{x}_2$. If, for example, $\texttt{i}_{12} = 0$, then $\texttt{x}_2$ gets the value 17 and the algorithm adds $\texttt{x}_2$ to the worklist. When it reevaluates the ϕ-function, it sets $\texttt{x}_1 = 17 \wedge 17 = 17$.

Consider what would happen if $\texttt{i}_{12} = 2$, instead. Then, when SSCP evaluates $\texttt{x}_1 + \texttt{i}_{12}$ it sets $\texttt{x}_2 = 19$. Next, it reevaluates $\texttt{x}_1 = 17 \wedge 19 = \bot$. This $\bot$, in turn, propagates to $\texttt{x}_2$, proving x nonconstant in the loop.

The Value of SSA Form

The use of SSA form in SSCP leads to a simple and efficient algorithm. To see this point, consider a classic data-flow approach to the problem. It would create a set CONSTANTSIN at the top of each block and a set CONSTANTSOUT at the end of each block. CONSTANTSIN and CONSTANTSOUT would hold $\langle variable, value \rangle$ pairs.

For a block b, the compiler could compute CONSTANTSIN(b) as a pairwise intersection of CONSTANTSOUT(p), taken over every $p \in preds(b)$. All the values for a single name would be intersected using the same meet function as in SSCP. To derive CONSTANTSOUT(b) from CONSTANTSIN(b) the compiler could apply a version of LVN extended to handle $\bot$ and $\top$. An iterative fixed-point algorithm would halt when the sets stopped changing.

This sketch oversimplifies the algorithm. This formulation lacks a unique fixed point, so the results depend on the order in which the blocks are processed. It also lacks the properties that let the iterative algorithm converge quickly. Solvers may run in $\mathbf{O}(n^2)$ time, or worse.

By contrast, SSCP is a simple iterative fixed-point algorithm operating on a sparse graph and particularly shallow lattice. It has the same complication with interpreting each operation over the known constant values, but it interprets single operations rather than whole blocks. It has an easily understood time bound. In this case, use of SSA form leads directly to a simple, efficient, sparse method for global constant propagation.

SECTION REVIEW

SSA form encodes information about both data flow and control flow in a conceptually simple intermediate form. This section focused on the algorithms to translate code into and out of *semipruned SSA form*. The initial construction of SSA form is a two-step process. The first step inserts ϕ-functions into the code at join points where distinct definitions can converge. That algorithm relies on dominance frontiers for efficiency. The second step creates the SSA name space by adding subscripts to the original base names during a systematic traversal of the entire procedure.

Because processors do not directly implement ϕ-functions, the compiler must translate code out of SSA form before it can execute. Transformation of the code while in SSA form can complicate out-of-SSA translation. Section 9.3.5 examined both the "lost copy problem" and the "swap problem" and described approaches for handling them. Finally, Section 9.3.6 showed an algorithm for global constant propagation over the SSA-form.

REVIEW QUESTIONS

1. Maximal SSA form includes useless ϕ-functions that define nonlive values and redundant ϕ-functions that merge identical values (e.g., $x_8 \leftarrow \phi(x_7, x_7)$). Can semipruned SSA insert nonlive or redundant ϕ-functions? If so, how can the compiler eliminate them?
2. Assume that your compiler targets an ISA that implements `swap` r_1, r_2, which simultaneously performs $r_1 \leftarrow r_2$ and $r_2 \leftarrow r_1$. What impact could `swap` have on out-of-SSA translation?

9.4 INTERPROCEDURAL ANALYSIS

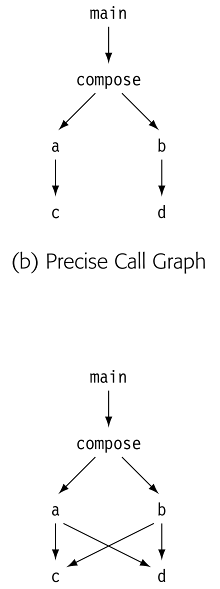
Procedure calls introduce two kinds of inefficiencies: (1) loss of knowledge in single-procedure analysis and optimization because of a call site; and (2) overhead introduced to implement the abstractions inherent in procedure calls. Interprocedural analysis was introduced to address the former problem. We saw, in Section 9.2.4, that the compiler can compute sets that summarize each call site's side effects. This section explores more complex issues in interprocedural analysis.

9.4.1 Call-Graph Construction

The first problem that the compiler must address in interprocedural analysis is the construction of a call graph. In the simplest case, in which

```
int compose( int f(), int g() ) {
  return f(g);
}
int a( int z() ) {
  return z();
}
int b( int z() ) {
  return z();
}
int c( ) {
  return 0;
}
int d( ) {
  return 1;
}
int main( int argc, char *argv[] ) {
  printf("(a,c) returns %d.\n",compose(a,c));
  printf("(b,d) returns %d,\n",compose(b,d));
}
```

(a) Example C Program



(b) Precise Call Graph

(c) Approximate Call Graph

■ **FIGURE 9.23** Building a Call Graph with Function-Valued Parameters.

every procedure call invokes a procedure named by a literal constant, as in `call fee(x, y, z)`, the problem is straightforward. The compiler creates a call-graph node for each procedure in the program and adds an edge to the call graph for each call site. This process takes time proportional to the number of procedures and the number of call sites in the program; in practice, the limiting factor will be the cost to locate the call sites.

Source language features can complicate call-graph construction. For example, consider the small C program shown in Fig. 9.23(a). Its precise call graph is shown in panel (b). The following subsections outline the language features that complicate call-graph construction.

Procedure-Valued Variables

If the program uses procedure-valued variables, the compiler must either assume that a call to a procedure-valued variable can invoke any procedure, or it must analyze the program to estimate the set of possible callees at each such call site. To perform this analysis, the compiler can construct the graph specified by the calls that use explicit literal constants. Next, it can track the propagation of functions as values around this subset of the call graph, adding edges as indicated.

The compiler can use a simple analog of global constant propagation to transfer function values from a procedure's entry to the call sites that use them, using set union as its meet operation.

In SSCP, initialize any function-valued formal parameters with known constant values. Actual parameters with the known values reveal where functions are passed through.

Once it has a set of procedures that might be passed to a procedure-valued parameter, the compiler must model the transmission of that parameter to individual call sites in the procedure. Most programming languages do not allow operations on a procedure-value, so this modeling can be both simple and effective (see the discussion of jump functions in Section 9.4.2).

Fig. 9.23 shows that a straightforward analysis may overestimate the set of call-graph edges. The code calls `compose` to compute `a(c)` and `b(d)`. A simple analysis, however, will conclude that the formal parameter `g` in `compose` can receive either `c` or `d`, and that, as a result, the program might compose any of `a(c)`, `a(d)`, `b(c)`, or `b(d)`, as shown in panel (c). To build the precise call graph, shown in panel (b), the compiler must track sets of parameters that are passed together, along the same path. The algorithm could then consider each set independently to derive the precise graph. Alternatively, it might tag each value with the path that the values travel and use the path information to avoid adding spurious edges such as (`a`,`d`) or (`b`,`c`).

Contextually Resolved Names

Some languages allow programmers to use names that are resolved by context. In object-oriented languages with an inheritance hierarchy, the binding of a method name to a specific implementation depends on the class of the receiver and the state of the inheritance hierarchy.

If the inheritance hierarchy and all the procedures are fixed at the time of analysis, then the compiler can use interprocedural analysis of the class structure to narrow the set of methods that can be invoked at any given call site. The call-graph constructor must include an edge from that call site to each procedure or method that might be invoked.

For a language that allows the program to import either executable code or new class definitions at runtime, the compiler must construct a conservative call graph that reflects the complete set of potential callees at each call site. One option is to have the compiler construct a single call-graph node to represent these unknown procedures and to endow that node with worst-case behavior, such as maximal MAYMOD and MAYREF sets. This strategy will ensure that other analyses have conservative approximations to the set of possible facts.

Dynamic linking, used in some operating systems to reduce virtual memory requirements, introduces similar complications. If the compiler cannot determine what code will execute, it cannot construct a complete call graph.

Analysis to resolve ambiguous calls can improve the precision of the call graph by reducing the number of spurious edges—edges for calls that cannot occur at runtime. Of equal or greater importance, any call site that can be resolved to a single callee can be implemented with a direct call; one with multiple callees may need a runtime lookup to dispatch the call (see Section 6.3.2). Runtime lookups to support dynamic dispatch can be much more expensive than a direct call.

Other Language Issues

In intraprocedural analysis, we assume that the control-flow graph has a single entry and a single exit; we add an artificial exit node if the procedure has multiple returns. The analogous problems arise in interprocedural analysis.

For example, JAVA has both initializers and finalizers. The JAVA virtual machine invokes a class initializer after it loads and verifies the class; it invokes an object initializer after it allocates space for the object but before it returns the object's hashcode. Thread start methods, finalizers, and destructors also have the property that they execute without an explicit call in the source program.

The call-graph builder must recognize and understand these procedures. It must connect them into the call graph in appropriate ways. The specific details will depend on the language definition and the analysis being performed. MAYMOD analysis, for example, might ignore them as irrelevant, while interprocedural constant propagation might need information from initialization and start methods.

9.4.2 Interprocedural Constant Propagation

Interprocedural constant propagation tracks known constant values of global variables and parameters as they propagate around the call graph, both through procedure bodies and across call-graph edges. The goal of interprocedural constant propagation is to discover places where a procedure always receives a known constant value or where a procedure always returns a known constant value. When the compiler finds such a constant, it can specialize the code to that value.

For the moment, we will restrict our attention to finding constant-valued formal parameters. The extension to global variables appears at the end of this section.

Conceptually, interprocedural constant propagation consists of three subproblems: discovering an initial set of constants, propagating known constant values around the call graph, and modeling transmission of values through procedures.

Discovering an Initial Set of Constants

The analyzer must identify, at each call site, which actual parameters have known constant values. A wide range of techniques are possible. The simplest method is to recognize literal constant values used as parameters. A more effective and expensive approach could use global constant propagation (e.g., SSCP from Section 9.3.6) to identify constant-valued parameters.

Propagating Known Constant Values Around the Call Graph

Given an initial set of constants, the analyzer propagates the constant values across call-graph edges and through the procedures from entry to each call site in the procedure. This portion of the analysis resembles the iterative data-flow algorithms from Section 9.2. The iterative algorithm will solve this problem, but it may require significantly more iterations than it would for simpler problems such as live variables or available expressions.

Modeling Transmission of Values Through Procedures

Each time the analyzer processes a call-graph node, it must determine how the constant values known at the procedure's entry affect the set of constant values known at each of the call sites in the procedure. To do so, it builds a small model for each actual parameter, called a *jump function*. At a call site s, we will denote the jump function for parameter a as $\mathcal{J}_s^a$.

Each call site s is represented with a vector of jump functions. If s has n parameters, the algorithm builds the vector $\mathcal{J}_s = \langle \mathcal{J}_s^a, \mathcal{J}_s^b, \mathcal{J}_s^c, \ldots, \mathcal{J}_s^n \rangle$, where a is the first formal parameter in the callee, b is the second, and so on. Each jump function, $\mathcal{J}_s^x$, relies on the values of some subset of the global variables and the formal parameters to the procedure p that contains s; we denote that set as *Support*($\mathcal{J}_s^x$).

For the moment, assume that $\mathcal{J}_s^x$ consists of an expression tree whose leaves are all global variables, formal parameters of the caller, or literal constants. We require that $\mathcal{J}_s^x$ return $\top$ if *Value*(y) is $\top$ for any $y \in$ *Support*($\mathcal{J}_s^x$).

The Algorithm

Fig. 9.24 shows a simple interprocedural constant propagation algorithm. It is similar to the SSCP algorithm presented in Section 9.3.6.

The algorithm associates a field *Value*(x) with each formal parameter x of each procedure p. (It assumes unique, or fully qualified, names for each formal parameter.) The first phase optimistically sets all the *Value* fields to $\top$. Next, it iterates over each actual parameter a at each call site s in the program, updates the *Value* field of a's corresponding formal parameter f to

```
// Phase 1: Initializations
Build all jump functions and Support mappings
Worklist ← $\emptyset$

for each procedure p in the program do
    for each formal parameter f of p do
        Value(f) ← $\top$                     // Optimistic initial value
        Worklist ← Worklist $\cup$ {f}

for each call site s in the program do
    for each formal parameter f that receives a value at s do
        Value(f) ← Value(f) $\wedge$ $\mathcal{J}_s^f$     // Initial constants factor into $\mathcal{J}_s^f$

// Phase 2: Iterate to a fixed point
while (Worklist $\neq \emptyset$) do
    pick parameter f from Worklist      // Pick an arbitrary parameter
    let p be the procedure declaring f

    // Update the Value of each parameter that depends on f
    for each call site s in p and parameter x such that f $\in$ Support($\mathcal{J}_s^x$) do
        t ← Value(x)
        Value(x) ← Value(x) $\wedge$ $\mathcal{J}_s^x$     // Compute new value
        if (Value(x) < t) then
            Worklist ← Worklist $\cup$ {x}

// Postprocess Value sets to produce CONSTANTS
for each procedure p do
    CONSTANTS(p) ← $\emptyset$
    for each formal parameter f of p do
        if (Value(f) = $\top$) then
            Value(f) ← $\bot$

        if (Value(f) $\neq \bot$) then
            CONSTANTS(p) ← CONSTANTS(p) $\cup$ { $\langle$f, Value(f)$\rangle$ }
```

■ **FIGURE 9.24** Iterative Interprocedural Constant Propagation Algorithm.

Value(f) $\wedge$ $\mathcal{J}_s^f$, and adds f to the worklist. This step factors the initial set of constants represented by the jump functions into the *Value* fields and sets the worklist to contain all of the formal parameters.

The second phase repeatedly selects a formal parameter from the worklist and propagates it. To propagate formal parameter f of procedure p, the analyzer finds each call site s in p and each formal parameter x (which corresponds to an actual parameter of call site s) such that $f \in$ *Support*($\mathcal{J}_s^x$). It evaluates $\mathcal{J}_s^x$ and combines it with *Value*(x). If *Value*(x) changes, it adds x to the worklist. The worklist should be implemented with a data structure, such as a sparse set, that does not allow duplicate members (see Section B.2.3).

The second phase terminates because each *Value* set can take on at most three values in the semilattice: $\top$, some c_i, and $\bot$. A variable x can only enter the worklist when its initial *Value* is computed or when its *Value* changes. Each variable x can appear on the worklist at most three times. Thus, the total number of changes is bounded and the iteration halts. After the second phase halts, a postprocessing step constructs the sets of constants known on entry to each procedure.

This algorithm relies on the same semilattice-based termination argument used for SSCP in Section 9.3.6.

Jump Function Implementation

Implementations of jump functions range from simple static approximations that do not change during analysis, through small parameterized models, to more complex schemes that perform extensive analysis at each jump-function evaluation. In any of these schemes, several principles hold. If the analyzer determines that parameter x at call site s is a known constant c, then $\mathcal{J}_s^x = c$ and $Support(\mathcal{J}_s^x) = \emptyset$. If $y \in Support(\mathcal{J}_s^x)$ and $Value(y) = \top$, then $\mathcal{J}_s^x = \top$. If the analyzer determines that the value of $\mathcal{J}_s^x$ cannot be determined, then $\mathcal{J}_s^x = \bot$.

For example, $Support(\mathcal{J}_s^x)$ might contain a value read from a file, so $\mathcal{J}_s^x = \bot$.

The analyzer can implement $\mathcal{J}_s^x$ in many ways. A simple implementation might only propagate a constant if the value enters the procedure as a formal parameter and passes, unchanged, to a parameter at a call site—that is, an actual parameter x is the SSA name of a formal parameter in the procedure that contains call site s. (Similar functionality can be obtained using REACHES information from Section 9.2.4.)

More complex schemes that find more constants are possible. The compiler could build expressions composed of SSA names of formal parameters and literal constants. The jump-function would then interpret the expression over the semilattice values of the SSA names and constants that it contains. To obtain even more precise results, the compiler could run the SSCP algorithm on demand to update the values of jump functions.

Extending the Algorithm

The algorithm shown in Fig. 9.24 only propagates constant-valued actual parameters forward along call-graph edges. We can extend it, in a natural way, to handle returned values and variables that are global to a procedure.

Just as the algorithm builds jump functions to model the flow of values from caller to callee, it can construct *return jump functions* to model the values returned from callee to caller. Return jump functions are particularly important for routines that initialize values, whether filling in a common block in FORTRAN or setting initial values for an object or class in JAVA. The algorithm can treat return jump functions in the same way that it handled ordinary jump functions; the one significant complication is that the implementation must avoid creating cycles of return jump functions that diverge (e.g., for a tail-recursive procedure).

To extend the algorithm to cover a larger class of variables, the compiler can extend the vector of jump functions in an appropriate way. Expanding the set of variables will increase the cost of analysis, but two factors mitigate the cost. First, in jump-function construction, the analyzer can notice that many of those variables do not have a value that can be modeled easily; it can map those variables onto a universal jump function that returns $\perp$ and avoid placing them on the worklist. Second, for the variables that might have constant values, the structure of the lattice ensures that they will be on the worklist at most twice. Thus, the algorithm should still run quickly.

SECTION REVIEW

Compilers perform interprocedural analysis to capture the behavior of all the procedures in the program and to bring that knowledge to bear on optimization within individual procedures. To perform interprocedural analysis, the compiler must model all of the code that it analyzes. A typical interprocedural problem requires the compiler to build a call graph (or some analog), to annotate it with information derived directly from the individual procedures, and to propagate that information around the graph.

The results of interprocedural information are applied directly in intraprocedural analysis and optimization. For example, MayMod and MayRef sets can be used to mitigate the impact of a call site on global data-flow analyses or to avoid the necessity for ϕ-functions after a call site. The results of interprocedural constant propagation can be used to initialize a global algorithm, such as sparse conditional constant propagation (see Section 10.7.1).

REVIEW QUESTIONS

1. Call-graph construction has many similarities to interprocedural constant propagation. The call-graph algorithm can achieve good results with relatively simple jump functions. What features could a language designer add that might necessitate more complex jump functions in the call-graph constructor?
2. How might the analyzer incorporate MayMod information into interprocedural constant propagation? What effect would you expect it to have?

9.5 ADVANCED TOPICS

Section 9.2 focused on iterative data-flow analysis. It emphasized the iterative approach because it is simple, robust, and efficient. Other approaches to data-flow analysis tend to rely heavily on structural properties of the underlying graph. Section 9.5.1 discusses flow-graph reducibility—a critical property for most of the structural algorithms.

Section 9.5.2 revisits the iterative dominance framework from Section 9.2.1. The simplicity of that framework makes it attractive; however, more specialized and complex algorithms have significantly lower asymptotic complexities. In Section 9.5.2, we introduce a set of data structures that make the simple iterative technique competitive with the fast dominator algorithms for flow graphs of up to several thousand nodes.

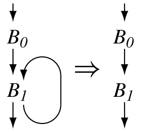
9.5.1 Structural Data-Flow Analysis and Reducibility

Chapters 8 and 9 present an iterative formulation of data-flow analysis. The iterative algorithm works, in general, on any set of well-formed equations on any graph. Other data-flow algorithms exist; many of these work by deriving a simple model of the control-flow structure of the code being analyzed and using that model to solve the equations. Often, that model is built by finding a sequence of transformations to the CFG that reduce its complexity—by combining nodes or edges in carefully defined ways. This graph-reduction process lies at the heart of almost every data-flow algorithm *except* the iterative algorithm.

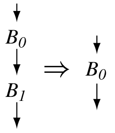
These *structural* data-flow algorithms use a small set of transformations, each of which selects a subgraph and replaces it by a single node to represent the subgraph. This creates a series of derived graphs in which each graph differs from its predecessor in the series by the effect of applying a single transformation. As the analyzer transforms the graph, it computes data-flow sets for the new representer nodes in each successive derived graph. These sets summarize the replaced subgraph's effects. The transformations reduce well-behaved graphs to a single node. The algorithm then reverses the process, going from the final derived graph, with its single node, back to the original flow graph. As it expands the graph back to its original form, the analyzer computes the final data-flow sets for each node.

Reducible graph
A flow graph is *reducible* if the two transformations, T_1 and T_2, will reduce it to a single node. If that process fails, the graph is *irreducible*.

In essence, the reduction phase gathers information from the entire graph and consolidates it, while the expansion phase propagates the effects in the consolidated set back out to the nodes of the original graph. Any graph for which such a reduction sequence succeeds is deemed *reducible*. If the graph cannot be reduced to a single node, it is *irreducible*.



$T_1(B_1)$



$T_2(B_0,B_1)$

To demonstrate reducibility, we can use the two graph transformations, called T_1 and T_2, shown in the margin. These same transformations form the basis for a classic data-flow algorithm. T_1 removes a self loop, which is an edge that runs from a node back to itself. The drawing in the margin shows T_1 applied to B_1, denoted $T_1(B_1)$. T_2 folds a node B_1 that has exactly one predecessor B_0 back into B_0; it removes the edge $\langle B_0, B_1 \rangle$, and makes B_0 the source of any edges that originally left B_1. If this leaves multiple edges from B_0 to some other node n, it consolidates those edges. The drawing in the margin shows $T_2(B_0, B_1)$.

Any graph that can be transformed, or *reduced*, to a single node by repeated application of T_1 and T_2 is deemed reducible. To understand how this works, consider the CFG from our continuing example. Fig. 9.25(a) shows one sequence of applications of T_1 and T_2 that reduces the CFG to a single-node graph. The sequence applies T_2 until no more opportunities exist: $T_2(B_1, B_2)$, $T_2(B_5, B_6)$, $T_2(B_5, B_8)$, $T_2(B_5, B_7)$, $T_2(B_1, B_5)$, and $T_2(B_1, B_3)$. Next, it uses $T_1(B_1)$ to remove the loop. Finally, it applies $T_2(B_0, B_1)$ and $T_2(B_0, B_4)$ to reduce the graph to a single node. This sequence proves that the graph is reducible.

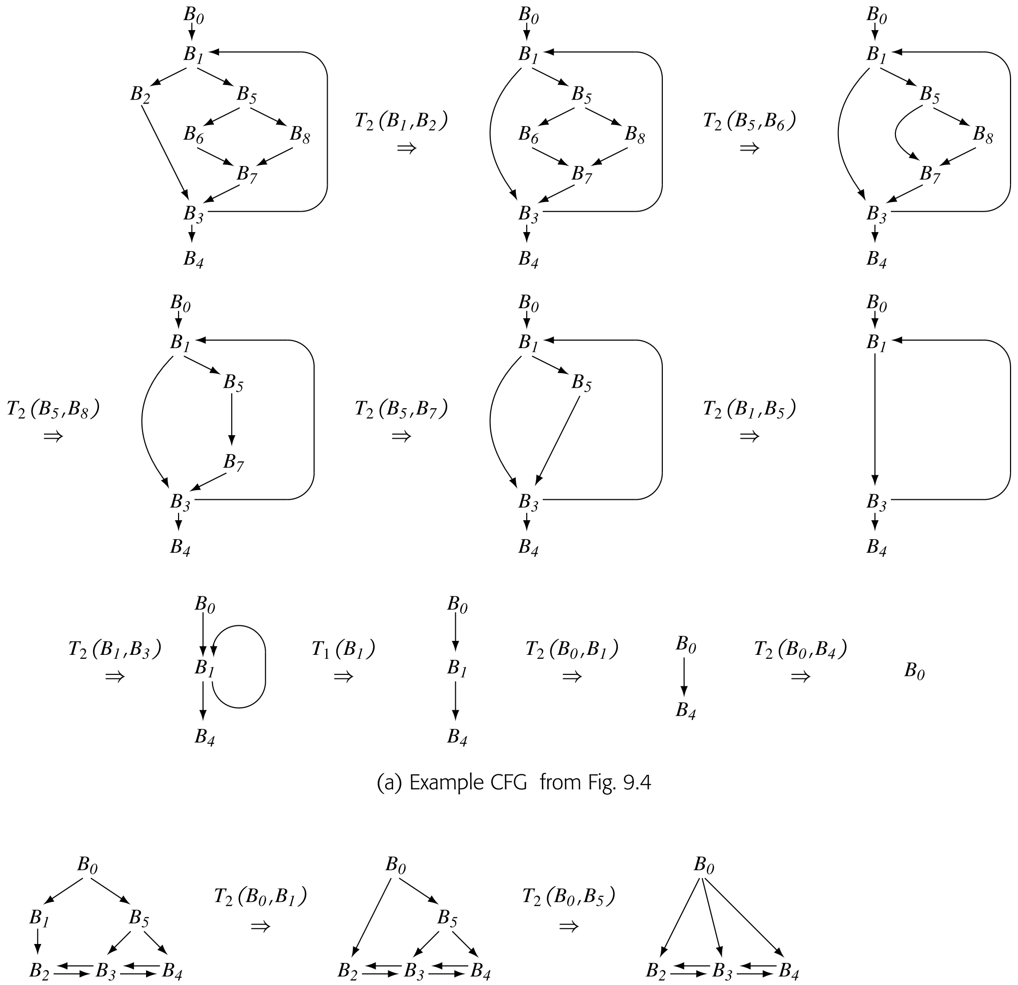
Other application orders also reduce the graph. For example, starting with $T_2(B_1, B_5)$ leads to a different transformation sequence. T_1 and T_2 have the finite Church-Rosser property, which ensures that the final result is independent of the order of application and that the sequence terminates. Thus, the analyzer can find places in the graph where T_1 or T_2 applies and use them opportunistically.

Fig. 9.25(b) shows what can happen when we apply T_1 and T_2 to a graph with multiple-entry loops. The analyzer uses $T_2(B_0, B_1)$ followed by $T_2(B_0, B_5)$. At that point, however, no remaining node or pair of nodes is a candidate for either T_1 or T_2. Thus, the analyzer cannot reduce the graph any further. (No other order will work either.) The graph cannot be reduced to a single node; it is irreducible.

Many other tests for graph reducibility exist. One fast and simple test is to apply the iterative DOM framework to the graph, using an RPO traversal order. If the calculation needs more than two iterations over a graph, that graph is irreducible.

The failure of T_1 and T_2 to reduce this graph arises from a fundamental property of the graph. The graph is irreducible because it contains a loop, or cycle, that has edges that enter it at different nodes. In terms of the source language, the program that generated the graph has a loop with multiple entries. We can see this property in the graph; consider the cycle formed by B_2 and B_3. It has edges entering it from B_1, B_4, and B_5. Similarly, the cycle formed by B_3 and B_4 has edges that enter it from B_2 and B_5.

Irreducibility poses a serious problem for algorithms built on transformations like T_1 and T_2. If the algorithm cannot reduce the graph to a single-node, then the method must either report failure, modify the graph



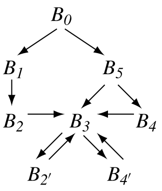
(a) Example CFG from Fig. 9.4

(b) An Irreducible Graph

■ **FIGURE 9.25** Reduction Sequences.

by splitting one or more nodes, or use an iterative approach to solve the system on the partially reduced graph. In general, structural algorithms for data-flow analysis only work on reducible graphs. The iterative algorithm, by contrast, works correctly, albeit more slowly, on an irreducible graph.

To transform an irreducible graph to a reducible graph, the analyzer can split one or more nodes. The simplest split for the example graph from Fig. 9.25(b) is shown in the margin. The transformation has cloned B_2 and B_4 to create $B_{2'}$ and $B_{4'}$, respectively. The analyzer then retargets the edges



Irreducible Graph After Node Splitting

(B_3, B_2) and (B_3, B_4) to form a complex loop, $\{B_3, B_{2'}, B_{4'}\}$. The new loop has a single entry, through B_3.

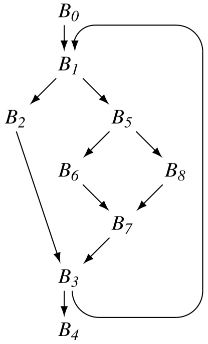
This transformation creates a reducible graph that executes the same sequence of operations as the original graph. Paths that, in the original graph, entered B_3 from either B_2 or B_4 now execute as prologs to the loop $\{B_3, B_{2'}, B_{4'}\}$. Both B_2 and B_4 have unique predecessors in the new graph. B_3 has multiple predecessors, but it is the sole entry to the loop and the loop is reducible. Thus, node splitting produced a reducible graph, at the cost of cloning two nodes.

In the reverse CFG, the `break` becomes a second entry to the cyclic region.

Both folklore and published studies suggest that irreducible graphs rarely arise in global data-flow analysis. The rise of structured programming in the 1970s made programmers much less likely to use arbitrary transfers of control, like a `goto` statement. Structured loop constructs, such as `do`, `for`, `while`, and `until` loops, cannot produce irreducible graphs. However, transferring control out of a loop (for example, C's `break` statement) creates a CFG that is irreducible to a backward analysis. Similarly, irreducible graphs may arise more often in interprocedural analysis due to mutually recursive subroutines. For example, the call graph of a recursive-descent parser is likely to have irreducible subgraphs. Fortunately, an iterative analyzer can handle irreducible graphs correctly and efficiently.

A simple way to avoid worst case behavior from an irreducible graph in an iterative analyzer is to compute two traversal orders, one based on the treewalk that traverses siblings left-to-right and another based on a right-to-left traversal. Alternating between these two orders on successive passes will improve behavior on worst-case irreducible graphs.

9.5.2 Speeding up the Iterative Dominance Framework



Example CFG

The iterative framework for computing dominance is particularly simple. Where most data-flow problems have equations involving several sets, the equations for DOM involve computing a pairwise intersection over DOM sets and adding a single element to those sets. The simple nature of these equations presents an opportunity; we can use a sparse data-structure to improve the speed of the DOM calculation.

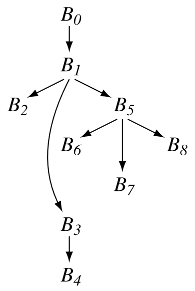
The iterative DOM framework described in Section 9.2.1 stores a full DOM set at each node. The compiler can achieve the same result by storing just the immediate dominator, or IDOM, at each node and solving for IDOM. The compiler can easily recreate DOM(n) when needed. Since IDOM is a singleton set, the implementation can be quite efficient.

```
for all nodes b do    // initialize the dominators array
    IDoms[b] ← Undefined
IDoms[$b_0$] ← $b_0$
Changed ← true
while (Changed) do
    Changed ← false
    for all nodes, b, in reverse postorder (except root) do
        NewIDom ← first (processed) predecessor of b    // pick one
        for all other predecessors, p, of b do
            if IDoms[p] $\neq$ Undefined then    // i.e., Doms[p] already calculated
                NewIdom ← Intersect(p, NewIdom)
        if IDoms[b] $\neq$ NewIdom then
            IDoms[b] ← NewIdom
            Changed ← true

Intersect(i, j)
    finger1 ← i
    finger2 ← j
    while (finger1 $\neq$ finger2)
        while (RPO(finger1) > RPO(finger2)) do
            finger1 = IDoms[finger1]
        while (RPO(finger2) > RPO(finger1)) do
            finger2 = IDoms[finger2]
    return finger1
```

■ **FIGURE 9.26** The Modified Iterative Dominator Algorithm.

Recall our example CFG from Section 9.2.1, repeated in the margins along with its dominator tree. Its IDOM sets are as follows:

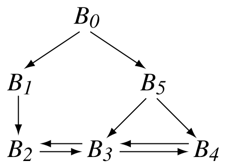
	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
IDOM(n)	–	0	1	1	3	1	5	5	5



Example's Dominator Tree

Notice that the dominator tree and the IDOMs are isomorphic. IDOM(b) is just b's predecessor in the dominator tree. The root of the dominator tree has no predecessor; accordingly, its IDOM set is undefined.

The compiler can read a graph's DOM sets from its dominator tree. For a node n, its DOM set is just the set of nodes that lie on the path from n to the root of the dominator tree, inclusive of the end points. In the example, the dominator-tree path from B_7 to B_0 consists of (B_7, B_5, B_1, B_0), which matches DOM(B_7) from Section 9.2.1.



(a) An Irreducible Graph

	B_0	B_1	B_2	B_3	B_4	B_5
RPO(n)	0	1	5	4	3	2

(b) A Worst-Case RPO

IDOM(n)						
	B_0	B_1	B_2	B_3	B_4	B_5
0	0	–	–	–	–	–
1	0	0	0	5	5	0
2	0	0	0	0	5	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0

(c) Progress of the IDOM Computation

■ **FIGURE 9.27** Computing Dominators on an Irreducible Graph.

Thus, the compiler can use the IDOM sets as a proxy for the DOM sets, provided that it can initialize and intersect the sets efficiently. A small modification to the iterative algorithm can simplify initialization. Intersection requires a more subtle approach, shown in Fig. 9.26. The critical procedure, *Intersect*, relies on two observations:

1. When the algorithm walks the path from a node to the root to recreate a DOM set, it encounters the nodes in a consistent order. The intersection of two DOM sets is simply the common suffix of the labels on the paths from the nodes to the root.
2. The algorithm needs to recognize the common suffix. It starts at the two nodes whose sets are being intersected, *i* and *j*, and walks upward in the dominator tree from each of them toward the root. If we name the nodes by their RPO numbers, then a simple comparison will let the algorithm discover the nearest common ancestor—the IDOM of *i* and *j*.

Intersect is a variant of the classic "two finger" algorithm. It uses two pointers to trace paths upward through the tree. When they agree, they both point to the node representing the result of the intersection.

Fig. 9.26 shows a reformulated iterative algorithm for IDOM. It keeps the IDOM information in an array, *IDoms*. It initializes the IDOM entry for the root, b_0, to itself to simplify the rest of the algorithm. It processes the nodes in reverse postorder. In computing intersections, it ignores predecessors whose IDOMs have not yet been computed.

To see how the algorithm operates, consider the irreducible graph in Fig. 9.27(a). Panel (b) shows an RPO for this graph that illustrates the problems caused by irreducibility. Using this order, the algorithm miscomputes the IDOMs of B_3, and B_4 in the first iteration. It takes two iterations for the algorithm to correct those IDoms, and a final iteration to recognize that the IDoms have stopped changing.

This improved algorithm runs quickly. It has a small memory footprint. On any reducible graph, it halts in two passes: the first pass computes the correct IDom sets and the second pass confirms that no changes occur. An irreducible graph will take more than two passes. In fact, the algorithm provides a rapid test for reducibility—if any IDom entry changes in the second pass, the graph is irreducible.

9.6 SUMMARY AND PERSPECTIVE

Most optimization tailors general-case code to the specific context that occurs in the compiled code. The compiler's ability to tailor code is often limited by its lack of knowledge about the program's range of runtime behaviors.

Data-flow analysis allows the compiler to model the runtime behavior of a program at compile time and to draw important, specific knowledge from these models. Many data-flow problems have been proposed; this chapter presented several of them. Many of those problems have properties that lead to efficient analyses.

SSA form is both an intermediate form and a tool for analysis. It encodes both data-flow information and control-dependence information into the name space of the program. Using SSA form as the basis for an algorithm has three potential benefits. It can lead to more precise analysis, because SSA incorporates control-flow information. It can lead to more efficient algorithms, because SSA is a sparse representation for the underlying data-flow information. It can lead to simpler formulations of the underlying optimization (see Section 10.7.2). These advantages have led both researchers and practitioners to adopt SSA form as a definitive representation in modern compilers.

CHAPTER NOTES

Credit for the first data-flow analysis is usually given to Vyssotsky at Bell Labs in the early 1960s [351]. Lois Haibt's work, in the original FORTRAN compiler, predates Vyssotsky. Her phase of the compiler built a control-flow graph and performed a Markov-style analysis over the CFG to estimate execution frequencies [27].

Iterative data-flow analysis has a long history in the literature. Among the seminal papers on this topic are Kildall's 1973 paper [234], work by Hecht and Ullman [197], and two papers by Kam and Ullman [221,222]. The treatment in this chapter follows Kam & Ullman.

This chapter focuses on iterative data-flow analysis. Many other algorithms for solving data-flow problems have been proposed [229]. The interested reader should explore the structural techniques, including interval analysis [18,19,68]; T_1-T_2 analysis [196,348]; the Graham-Wegman algorithm [178,179]; the balanced-tree, path-compression algorithm [342,343]; graph grammars [230]; and the partitioned-variable technique [371]. The alternating-direction iterative method mentioned at the end of Section 9.5.1 is due to Harvey [109].

Dominance has a long history in the literature. Prosser introduced dominance in 1959 but gave no algorithm to compute dominators [300]. Lowry and Medlock describe the algorithm used in their compiler [260]; it takes at least $\mathsf{O}(N^2)$ time, where N is the number of statements in the procedure. Several authors developed faster algorithms based on removing nodes from the CFG [4,9,301]. Tarjan proposed an $\mathsf{O}(N \log N + E)$ algorithm based on depth-first search and union find [341]. Lengauer and Tarjan improved this time bound [252], as did others [24,67,190]. The data-flow formulation for dominators is taken from Allen [13,18]. The fast data structures for iterative dominance are due to Harvey [110]. The algorithm in Fig. 9.10 is from Ferrante, Ottenstein, and Warren [155].

The SSA construction is based on the seminal work by Cytron et al. [120]. That work builds on work by Shapiro and Saint [323]; by Reif [305, 344]; and by Ferrante, Ottenstein, and Warren [155]. The algorithm in Section 9.3.3 builds semipruned SSA [55]. Briggs et al. describe the details of the renaming algorithm and the ad-hoc approach to out-of-SSA translation [56]. The unified approach to out-of-SSA translation is due to Boissinot et al. [51]. The complications introduced by critical edges have long been recognized in the literature of optimization [139,141,144,236,312]; it should not be surprising that they also arise in the translation from SSA back into executable code. The sparse simple constant algorithm, SSCP, is due to Reif and Lewis [306]. Wegman and Zadeck reformulate SSCP to use SSA form [358,359].

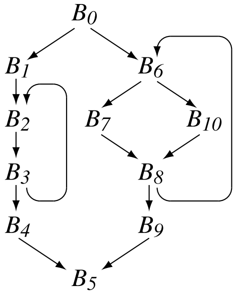
The IBM PL/I optimizing compiler was one of the earliest systems to perform interprocedural data-flow analysis [334]. Call-graph construction is heavily language dependent: Ryder looked at the problems that arise in Fortran [314], C [272], and JAVA [372]. Shivers wrote the classic paper on control-flow analysis in Scheme-like languages [325].

Early work on side-effect analysis focused more on defining the problems than on their fast solution [35,37]. Cooper and Kennedy developed simple frameworks for MAYMOD and MAYREF that led to fast algorithms for these problems [112,113]. The interprocedural constant propagation algorithm is from Torczon's thesis and subsequent papers [74,182,271]; both Cytron and Wegman suggested other approaches to the problem [121,359]. Burke and Torczon [70] formulated an analysis that determines which modules in a large program must be recompiled in response to a change in a program's interprocedural information. Pointer analysis is inherently interprocedural; a growing body of literature describes that problem [84,87,123,134,149, 202,203,209,247,322,360,363]. Ayers, Gottlieb, and Schooler described a practical system that analyzed and optimized a subset of the entire program [26].

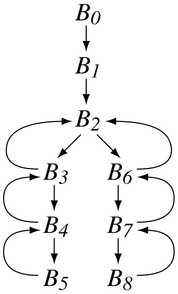
EXERCISES

Section 9.2

1. In live analysis, the equations initialize the LIVEOUT set of each block to ∅. Are other initializations possible? Do they change the result of the analysis? Justify your answer.
2. In live analysis, how should the compiler treat a block containing a procedure call? What should the block's UEVAR set contain? What should its VARKILL set contain?
3. For each of the following control-flow graphs:

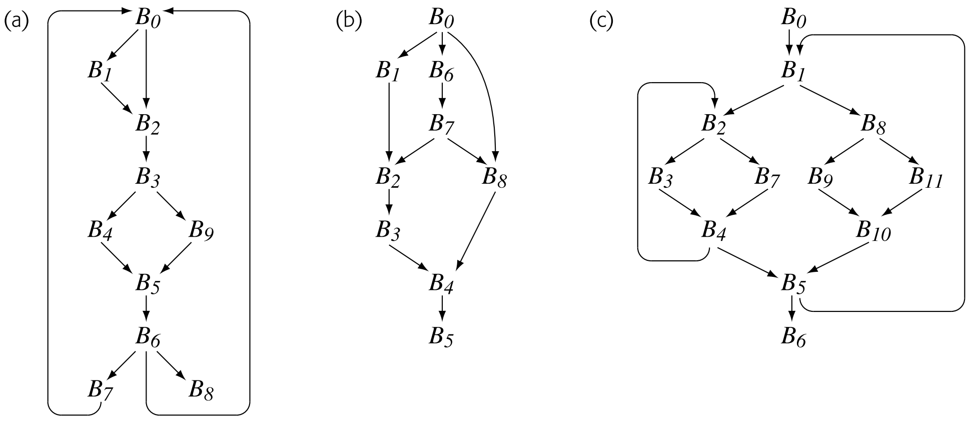


(a) Multiple Loops



(b) Doubled Loop Body

   a. Compute reverse postorder numberings for the CFG and the reverse CFG.
   b. Compute reverse preorder on the CFG.
   c. Is reverse preorder on the CFG equivalent to postorder on the reverse CFG?



■ **FIGURE 9.28** Control-Flow Graphs for Exercise 4.

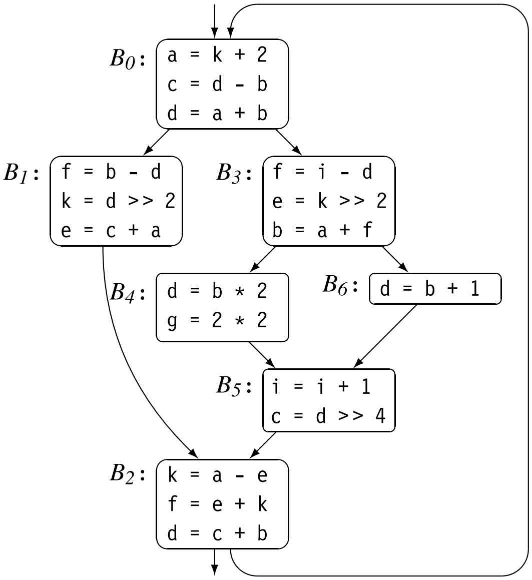
Section 9.3

4. Consider the three control-flow graphs shown in Fig. 9.28.

   a. Compute the dominator trees for CFGs (a), (b), and (c).

   b. Compute the dominance frontiers for nodes 3 and 5 of (a), nodes 4 and 5 of (b), and nodes 2 and 10 of (c).

5. Translate the code in the CFG shown below into SSA form. Show only the final results, after both ϕ-insertion and renaming.



B_0:
```
a = k + 2
c = d - b
d = a + b
```

B_1:
```
f = b - d
k = d >> 2
e = c + a
```

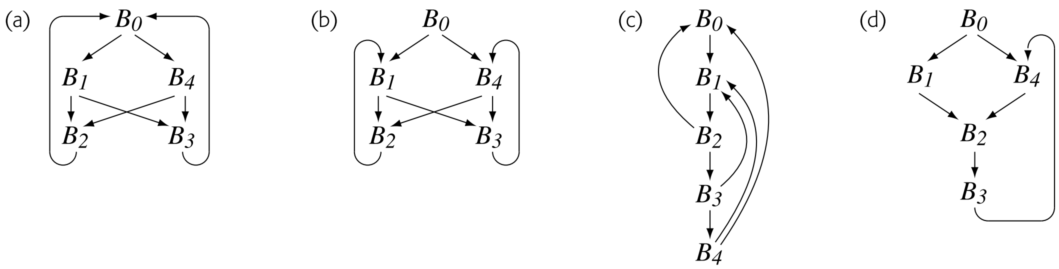
B_3:
```
f = i - d
e = k >> 2
b = a + f
```

B_4:
```
d = b * 2
g = 2 * 2
```

B_6:
```
d = b + 1
```

B_5:
```
i = i + 1
c = d >> 4
```

B_2:
```
k = a - e
f = e + k
d = c + b
```



■ **FIGURE 9.29** Control-Flow Graphs for Exercise 10.

6. Given an assignment to some variable v in block b, consider the set of blocks that need a ϕ-function as a result. The algorithm in Fig. 9.11 inserts a ϕ-function in each block in DF(b). It then adds each of those blocks to the worklist; they, in turn, may add more blocks to the worklist. Call the set of all these blocks $\text{DF}^+(b)$. We can define $\text{DF}^+(b)$ as the limit of the sequence:

The algorithm should only add a block to the worklist once.

$$\begin{aligned}
\text{DF}_1(b) &= \text{DF}(b)\\
\text{DF}_2(b) &= \text{DF}_1(b) \cup_{x\in \text{DF}_1(b)} \text{DF}_1(x)\\
\text{DF}_3(b) &= \text{DF}_2(b) \cup_{x\in \text{DF}_2(b)} \text{DF}_2(x)\\
&\;\;\ldots\\
\text{DF}_i(b) &= \text{DF}_{i\text{-}1}(b) \cup_{x\in \text{DF}_{i-1}(b)} \text{DF}_{i\text{-}1}(x)
\end{aligned}$$

Using these extended sets, $\text{DF}^+(b)$, leads to a simpler algorithm for inserting ϕ-functions.

a. Develop an algorithm to compute $\text{DF}^+(b)$.

b. Develop an algorithm to insert ϕ-functions using the DF^+ sets computed in part (a).

c. Compare the overall cost of your algorithm, including the computation of DF^+ sets, to the cost of the ϕ-insertion algorithm given in Section 9.3.3.

7. The maximal SSA construction is both simple and intuitive. However, it can insert many more ϕ-functions than the semipruned algorithm. In particular, it can insert both redundant ϕ-functions ($\texttt{x}_\texttt{i} \leftarrow \phi(\texttt{x}_\texttt{j},\texttt{x}_\texttt{j})$) and dead ϕ-functions—functions whose results are never used.

a. Propose a method to detect and remove the extra ϕ-functions that the maximal construction inserts.

b. Can your method reduce the set of ϕ-functions to just those that the semipruned construction inserts?

c. Contrast the asymptotic complexity of your method against that of the semipruned construction.

8. Apply the unified out-of-SSA translation scheme to the example code for the lost-copy problem, shown in Fig. 9.18(a).

9. Apply the unified out-of-SSA translation scheme to the example code for the swap problem, shown in Fig. 9.19(a).

Section 9.5

10. For each of the control-flow graphs shown in Fig. 9.29, show whether or not it is reducible.

    (**Hint:** use a sequence of T_1 and T_2 to show that the graph is reducible. If no such sequence exists, it is irreducible.)

11. Prove that the following definition of a reducible graph is equivalent to the definition that uses the transformations T_1 and T_2: "A graph G is reducible if and only if for each cycle in G, there exists a node n in the cycle with the property that n dominates every node in that cycle."

Chapter 10

Scalar Optimization

ABSTRACT

An optimizing compiler improves the quality of the code that it generates by applying transformations to rewrite the code. "Quality" might be measured in runtime speed, in code size, or in more complex metrics, such as the energy expended by the processor at runtime. This chapter builds on the introduction to optimization provided in Chapter 8 and the material on static analysis in Chapter 9 to focus on optimization of the code for a single thread of control—so-called scalar optimization. The chapter introduces a selection of machine-independent scalar transformations that address a variety of inefficiencies in compiled code.

KEYWORDS

Optimization, Transformation, Machine Independent, Machine Dependent, Redundancy, Dead Code, Constant Propagation

10.1 INTRODUCTION

A compiler's optimizer analyzes and transforms the IR form of the code in an attempt to improve that code's performance. It uses static analyses (see Chapter 9) to discover opportunities for transformations and to prove their safety. It then transforms the code, or rewrites it, in ways that are expected to produce better code in the compiler's back end.

Code optimization has a history that is as long as the history of compilers. The first FORTRAN compiler included careful optimization with the intent to provide performance that rivaled hand-written assembly code. Since that first optimizing compiler in the late 1950s, the literature on optimization has grown to include thousands of papers that describe analyses and transformations. Deciding which transformations to use and selecting an order in which to apply them remains one of the most daunting decisions that a compiler writer faces.

Scalar optimization
code improvement techniques that focus on a single thread of control

This chapter focuses on *scalar optimization*, that is, optimization of code along a single thread of control. It identifies five key sources of inefficiency in compiled code and presents optimizations that help to remove those inefficiencies. The chapter is organized around these five effects; we expect that

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00016-4

a compiler writer choosing optimizations might use the same organizational scheme.

Conceptual Roadmap

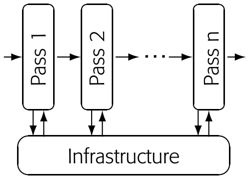
Compiler-based optimization is the process of analyzing the code to determine its properties and using the results of that analysis to rewrite the code into a more efficient or more effective form. Such improvement can be measured in many ways, including decreased running time, smaller code size, or lower processor energy use at runtime. Every compiler has some set of input programs for which it produces highly efficient code. A good optimizer should make that performance available on a much larger set of inputs. The optimizer should be robust, that is, small changes in the input should not produce wild changes in performance.

Machine independent
A transformation that improves code on most target machines is considered *machine independent*.

Machine dependent
A transformation that relies on specific knowledge of the target processor is considered *machine dependent*.

An optimizer achieves these goals through two primary mechanisms. It eliminates unnecessary overhead introduced to support programming language abstractions and it matches the needs of the resulting program to the available hardware and software resources of the target machine. In the broadest sense, transformations can be classified as either *machine independent* or *machine dependent*. For example, replacing a redundant computation with a reuse of the previously computed value is usually faster than recomputing the value; thus, redundancy elimination is considered machine independent. By contrast, implementing a character string copy operation with the "scatter-gather" hardware on a vector processor is clearly *machine dependent*. Rewriting that copy operation with a call to the hand-optimized system routine `memmove` might be more broadly applicable.

Overview



Most optimizers are built as a series of passes that share common infrastructure for analysis and manipulation of the IR, as shown in the margin. Each pass takes code in IR form as its input. Each pass produces a rewritten version of the IR code as its output. This structure breaks the implementation into smaller pieces and avoids some of the complexity that arises in large, monolithic programs. It allows the passes to be built and tested independently, which simplifies development, testing, and maintenance. It creates a natural way for the compiler to provide different levels of optimization; each level specifies a set of passes to run. The pass structure allows the compiler writer to run some passes multiple times, if desirable. In practice, some passes should run once, while others might run several times at different points in the sequence.

In the design of an optimizer, the selection of transformations and the ordering of those transformations play a critical role in determining the overall

OPTIMIZATION SEQUENCES

The choice of specific transformations and the order of their application have a strong impact on the effectiveness of an optimizer. To make the problem harder, individual transformations have overlapping effects (e.g., local value numbering versus superlocal value numbering) and different programs have different sets of inefficiencies.

Equally difficult, transformations that address different effects interact with one another. A given transformation can create opportunities for other transformations. Symmetrically, a given transformation can obscure or eliminate opportunities for other transformations.

Classic optimizing compilers provide several levels of optimization (e.g., `-O`, `-O1`, `-O2`, …) as one way of providing the end user with multiple sequences that they can try. Research has looked at techniques to derive custom sequences for specific application codes, selecting both a set of transformations and an order of application (see also Section 10.7.3).

effectiveness of the optimizer. The selection of transformations determines what specific inefficiencies in the IR program the optimizer discovers and how it rewrites the code to reduce those inefficiencies. The order in which the compiler applies the transformations determines how the passes interact.

For example, in the appropriate context (e.g., $r_2 > 0$ and $r_5 = 4$), an optimizer might replace `mult` $r_2, r_5 \Rightarrow r_{17}$ with `lshiftI` $r_2, 2 \Rightarrow r_{17}$. This change replaces a multicycle integer multiply with a single-cycle shift operation and reduces demand for registers. In most cases, this rewrite is profitable. If, however, the next transformation that the compiler applies uses commutativity to rearrange expressions, then replacing a multiply with a shift may foreclose opportunities. To the extent that a transformation makes later passes less effective, it may hurt overall code quality. Deferring the replacement of multiplies by shifts may avoid this problem; the context needed to prove safety and profitability for this rewrite is likely to survive the intervening passes.

Recall that multiplication is commutative but a shift is not.

The first hurdle in the design and construction of an optimizer is conceptual. The optimization literature describes hundreds of distinct algorithms to improve IR programs. The compiler writer must select a subset of these transformations to implement and apply. While reading the original papers may help with the implementation, it provides little insight for the decision process; most of the papers advocate the use of their own transformations.

Compiler writers need to understand both what inefficiencies arise in applications translated by their compilers and what impact those inefficiencies have on the application's performance. Given a set of specific flaws to address, they can then select specific transformations to address them. Many transformations, in fact, address multiple inefficiencies, so careful selection can reduce the number of passes needed. Since most optimizers are built with limited resources, the compiler writer can prioritize transformations by their expected impact on the final code.

The distinction between the categories can be unclear. We call a transformation machine independent if it deliberately ignores target machine considerations, such as its impact on register allocation.

As mentioned in the conceptual roadmap, transformations fall into two broad categories: machine-independent transformations and machine-dependent transformations. Examples of machine-independent transformations from earlier chapters include local value numbering (LVN), inline substitution, and constant propagation. Machine-dependent transformations often fall into the realm of code generation. Examples include peephole optimization (see Section 11.3), instruction scheduling, and register allocation. Other machine-dependent transformations fall into the realm of the optimizer. Examples include tree-height balancing, global code placement, and procedure placement. Some transformations resist classification; loop unrolling has diverse effects that include reduction of loop overhead (machine independent) and provision of more independent instructions to the scheduler (machine dependent).

Chapters 8 and 9 present a number of optimization techniques selected to illustrate specific points in those chapters. Chapters 11 to 13 focus on code generation, which is inherently machine dependent. This chapter presents a broad selection of transformations, most of which are machine-independent. The transformations are organized around the effect that they have on the final code. We will concern ourselves with five specific effects.

- *Eliminate Useless and Unreachable Code* The compiler can discover that an operation is either useless or unreachable. In most cases, removing such operations produces faster, smaller code.
- *Move Code* The compiler can move an operation to a point where it executes less often but produces the same answer. In most cases, code motion reduces runtime. It sometimes reduces code size.
- *Specialize a Computation* The compiler can specialize a code sequence to the context around it. Specialization reduces the cost of generic code sequences.
- *Eliminate a Redundant Computation* The compiler can prove that a value has already been computed and reuse the earlier value. In many cases, reuse costs less than recomputation.
- *Enable Other Transformations* The compiler can rewrite the code to expose new opportunities for other optimizations. Inline substitution, for example, enables many other optimizations.

OPTIMIZATION AS SOFTWARE ENGINEERING

Including an optimizer can simplify the design and implementation of a compiler. It simplifies the front end; the front end can generate general-purpose code and ignore special cases. It simplifies the back end; the back end can focus on mapping the IR version of the program to the target machine. Without an optimizer, both the front and back end must pay attention to opportunities to improve the code.

In a pass-structured optimizer, each pass contains a transformation and the analysis required to support it. In principle, each task that the optimizer performs can be implemented once. This provides a single point of control and lets the compiler writer implement complex functions once, rather than many times. For example, deleting an operation from the IR can be complicated. If the deleted operation leaves a basic block empty, except for the block-ending branch or jump, then the transformation should also delete the block and reconnect the block's predecessors to its successors, as appropriate. Building this functionality once simplifies implementation, understanding, and maintenance.

From a software engineering perspective, the pass structure makes sense. It focuses each pass on a single task. It creates a clear separation of concerns—value numbering ignores register pressure and register allocation ignores redundancy. It lets the compiler writer test passes independently and thoroughly, and it simplifies fault isolation.

This set of categories covers most machine-independent effects that the compiler can address. In practice, many transformations attack effects in more than one category. LVN, for example, eliminates redundant computations, specializes computations with known constant values, and uses algebraic identities to identify and remove some kinds of useless computations.

The rest of this chapter explores these five opportunities: dead code elimination, code motion, specialization, redundancy elimination, and enabling other transformations. It includes multiple transformations in each category. The Advanced Topics section presents an algorithm that combines constant propagation and dead code elimination, along with an algorithm for operator strength reduction.

10.2 DEAD CODE ELIMINATION

Sometimes, programs contain computations that have no externally visible effect. If the compiler can determine that a given operation does not affect the program's results, it can eliminate the operation. Most programmers do not write such code intentionally. However, it arises in most programs as the direct result of other optimizations or from naive translation in the compiler's front end.

Useless
An operation is *useless* if no operation uses its result, or if all uses of the result are, themselves dead.

An operation that does not change any value, such as a ← a + 0, is also *useless*.

Unreachable
An operation is *unreachable* if no valid control-flow path contains the operation.

Two distinct effects can make an operation eligible for removal as dead code. The operation can be *useless*, meaning that its result has no externally visible effect. Alternatively, the operation can be *unreachable*, meaning that it cannot execute. If an operation falls into either category, it can be eliminated. The term *dead code* is often used to mean either useless or unreachable code.

Removing useless or unreachable code shrinks the IR form of the code, which leads to a smaller executable program, to faster compilation, and, often, to faster execution. It may also increase the compiler's ability to improve the code. For example, unreachable code may have effects that show up in the results of static analysis and prevent the application of some transformations. In this case, removing the unreachable block may change the analysis results and allow further transformations (see, e.g., sparse conditional constant propagation in Section 10.7.1).

Some forms of redundancy elimination also remove useless code. As we saw in Section 8.4.1, LVN applies algebraic identities to simplify the code. Examples include a ← a + 0, b ← b x 1, and c ← max(c,c). Each of these simplifications eliminates a useless operation—by definition, an operation that, when removed, makes no difference in the program's externally visible behavior.

Because the algorithms in this section modify the program's control-flow graph (CFG), we carefully distinguish between the terms *branch*, as in an ILOC cbr, and *jump*, as in an ILOC jump. Close attention to this distinction will help the reader understand the algorithms.

10.2.1 Eliminating Useless Code

The classic algorithms for eliminating useless code operate in a manner similar to mark-sweep garbage collectors with the IR code as data (see Section 6.6.2). Like mark-sweep collectors, they perform two passes over the code. The first pass starts by clearing all the mark fields and marking "critical" operations as "useful." An operation is *critical* if it sets a return value for the procedure, it is an input/output statement, or it affects the value in a storage location that may be accessible from outside the current procedure. Examples of critical operations include a procedure's prolog and epilog code and the precall and postreturn sequences at calls. Next, the algorithm traces the operands of useful operations back to their definitions and marks

An operation can set a return value for a procedure in several ways, including assignment to a call-by-reference parameter or a global variable, assignment through an ambiguous pointer, or passing a return value via a return statement.

```
Mark( )
    WorkList ← Ø
    for each operation i do
        clear i's mark
        if i is critical then
            mark i
            WorkList ← WorkList ∪ {i}
    while (WorkList ≠ Ø) do
        remove an operation i from WorkList
            (assume i is x ← y op z)
        if def(y) is not marked then
            mark def(y)
            WorkList ← WorkList ∪ {def(y)}
        if def(z) is not marked then
            mark def(z)
            WorkList ← WorkList ∪ {def(z)}
        for each block b ∈ RDF(block(i)) do
            let x be the branch that ends b
            if x is unmarked then
                mark x
                WorkList ← WorkList ∪ {x}
```

(a) The *Mark* Phase

```
Sweep( )
    for each operation i do
        if i is unmarked then
            if i is a branch then
                rewrite i with a jump
                    to i's nearest marked
                    postdominator
            if i is not a jump then
                delete i
```

(b) The *Sweep* Phase

■ **FIGURE 10.1** Useless Code Elimination.

those operations as useful. This process continues, in a simple worklist iterative scheme, until no more operations can be marked as useful. The second pass walks the code and removes any operation not marked as useful.

Fig. 10.1 makes these ideas concrete. The algorithm, which we call *Dead*, assumes that the code is in SSA form. SSA simplifies the process because each use refers to a single definition. *Dead* consists of two passes. The first, called *Mark*, discovers the set of useful operations. The second, called *Sweep*, removes useless operations. *Mark* relies on control dependence, which is closely related to dominance frontiers (see Section 9.3.2).

The treatment of operations other than branches or jumps is straightforward. The marking phase determines whether an operation is useful. The sweep phase removes operations that have not been marked as useful.

The treatment of control-flow operations is more complex. Every jump is considered useful. Branches are considered useful only if the execution of a useful operation depends on their presence. As the marking phase discovers useful operations, it also marks the appropriate branches as useful. To map from a marked operation to the branches that it makes useful, the algorithm relies on the notion of control dependence.

Postdominance

In a CFG, *j postdominates i* if and only if every path from *i* to the exit node passes through *j*.

See also the definition of dominance on page 452.

The definition of control dependence relies on *postdominance*. In a CFG, node *j* postdominates node *i* if every path from *i* to the CFG's exit node passes through *j*. Using postdominance, we can define control dependence as follows: in a CFG, node *j* is control-dependent on node *i* if and only if

1. There exists a nonnull path from *i* to *j* such that *j* postdominates every node on the path after *i*. Once execution begins on this path, it must flow through *j* to reach the CFG's exit (from the definition of postdominance).
2. *j* does not strictly postdominate *i*. Another edge leaves *i* and control may flow along a path to a node not on the path to *j*. There must be a path beginning with this edge that leads to the CFG's exit without passing through *j*.

In other words, two or more edges leave block *i*. One or more edges lead to *j* and one or more edges do not. Thus, the decision made at the branch that ends block *i* can determine whether or not *j* executes. If an operation in *j* is useful, then the branch that ends *i* is also useful.

This notion of control dependence is captured precisely by the *reverse dominance frontier* of *j*, denoted RDF(*j*). Reverse dominance frontiers are simply dominance frontiers computed on the reverse CFG. When *Mark* marks an operation in block *b* as useful, it visits every block in *b*'s reverse dominance frontier and marks their block-ending branches as useful. As it marks these branches, it adds them to the worklist. It halts when that worklist is empty.

Sweep replaces any unmarked branch with a jump to the first postdominator block that contains a marked operation. If the branch is unmarked, then its successors, down to its immediate postdominator, contain no useful operations. (Otherwise, when those operations were marked, the branch would have been marked.) A similar argument applies if the immediate postdominator contains no marked operations. To find the nearest useful postdominator, the algorithm can walk up the postdominator tree until it finds a block that contains a useful operation. Since, by definition, the exit block is useful, this search must terminate.

After *Dead* runs, the code contains no useless computations. It may contain empty blocks, which can be removed by the next algorithm.

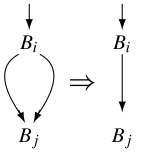
10.2.2 Eliminating Useless Control Flow

Optimization can rewrite the IR form of the program so that it has useless control flow. If the compiler includes optimizations that create useless control flow as a side effect, then it should include a pass that simplifies the CFG by eliminating useless control flow. This section presents a simple algorithm called *Clean* that handles this task.

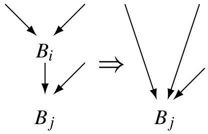
Clean operates directly on the procedure's CFG. It uses four transformations, shown in the margin, applied in the following order:

1. *Fold a Redundant Branch* If *Clean* finds a block that ends in a branch, and both sides of the branch target the same block, it replaces the branch with a jump. This situation arises as the result of other simplifications. For example, B_i might have had two successors, each with a jump to B_j. If another transformation had already emptied those blocks, then empty-block removal, discussed next, might produce the inital graph shown in the margin.
2. *Remove an Empty Block* If *Clean* finds a block that contains only a jump, it can merge the block into its successor. This situation arises when other passes remove all of the operations from a block B_i. Consider the initial graph shown in the margin. Since B_i has only one successor, B_j, the transformation retargets the edges that enter B_i to B_j and deletes B_i from B_j's set of predecessors. This transformation simplifies the graph. It should also speed up execution. In the original graph, the paths through B_i needed two control-flow operations to reach B_j. In the transformed graph, those paths use one operation to reach B_j.
3. *Combine Blocks* If *Clean* finds a block B_i that ends in a jump to B_j and B_j has only one predecessor, it can combine B_i and B_j, as shown in the margin. This situation can arise in several ways. Another transformation might eliminate other edges that entered B_j, or B_i and B_j might be the result of folding a redundant branch. In either case, the two blocks can be combined into a single block. This eliminates the jump at the end of B_i.
4. *Hoist a Branch* If *Clean* finds a block B_i that ends with a jump to an empty block B_j and B_j ends with a branch, *Clean* can replace the block-ending jump in B_i with a copy of the branch from B_j. The effect, is to hoist the branch into B_i, as shown in the margin. This situation arises when other passes eliminate the operations in B_j, leaving a jump to a branch. The rewritten code achieves the same effect with one fewer jump. It adds one edge to the CFG.

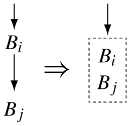
   Notice that B_i cannot be empty, or *Clean* would have removed it. Neither can it be B_j's sole predecessor; *Clean* would have combined B_1 and B_j. (After hoisting, B_j still has at least one predecessor.)



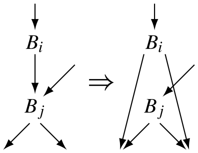
Fold a Redundant Branch



Removing an Empty Block



Combining Blocks



Hoisting a Branch

Some bookkeeping is required to implement these transformations. Some of the modifications are trivial. To fold a redundant branch in a program represented with ILOC and a graphical CFG, *Clean* simply overwrites the block-ending branch with a jump and adjusts the successor and predecessor

```
MakeAPass( )
  for each block i, in postorder do
    if i ends in a conditional branch then
      if both targets are identical then
        replace the branch with a jump                 /* case 1 */
    if i ends in a jump to j then
      if i is empty then
        replace transfers to i with transfers to j     /* case 2 */
      if j has only one predecessor then
        combine i and j                                /* case 3 */
      if j is empty and ends in a conditional branch then
        overwrite i's jump with a copy of j's branch   /* case 4 */

Clean( )
  while the CFG keeps changing do
    compute postorder
    MakeAPass( )
```

■ **FIGURE 10.2** The Algorithm for *Clean*.

lists of the blocks. Others are more difficult. Merging two blocks may involve allocating space for the merged block, copying the operations into the new block, adjusting the predecessor and successor lists of the new block and its neighbors in the CFG, and discarding the two original blocks.

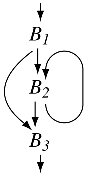
Many compilers and assemblers have included an ad-hoc pass that eliminates a jump to a jump or a jump to a branch. *Clean* achieves the same effect in a systematic way.

Clean applies these four transformations in a systematic fashion. It traverses the graph in postorder, so that B_i's successors are simplified before B_i, unless the successor lies along a back edge with respect to the postorder numbering. In that case, *Clean* will visit the predecessor before the successor. This traversal order is unavoidable in a cyclic graph. Simplifying successors before predecessors reduces the number of times that the implementation must move some edges.

In some situations, more than one of the transformations may apply. Careful analysis of the various cases leads to the order shown in Fig. 10.2, which corresponds to the order in which they were presented. The algorithm may apply multiple transformations to a block in a single visit.

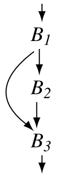
If the CFG contains back edges, then a pass of *Clean* may create additional opportunities—unprocessed successors along the back edges. These, in turn, may create other opportunities. Thus, *Clean* repeats the transformation sequence iteratively until the CFG stops changing. It must compute a new postorder numbering between calls to *MakeAPass* because each pass changes the underlying graph. Fig. 10.2 shows pseudocode for *Clean*.

Clean cannot, by itself, eliminate an empty loop. Consider the CFG shown in the margin. Assume that block B_2 is empty. None of *Clean*'s transformations can eliminate B_2 because the branch that ends B_2 is not redundant. B_2 does not end with a jump, so *Clean* cannot combine it with B_3. Its predecessor ends with a branch rather than a jump, so *Clean* can neither combine B_2 with B_1 nor fold its branch into B_1.



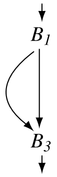
Original CFG

However, cooperation between *Clean* and *Dead* can eliminate the empty loop. *Dead* used control dependence to mark useful branches. If B_1 and B_3 contain useful operations, but B_2 does not, then the *Mark* pass in *Dead* will decide that the branch ending B_2 is not useful because $B_2 \notin \text{RDF}(B_3)$. Because the branch is useless, the code that computes the branch condition is also useless. Thus, *Dead* eliminates all of the operations in B_2 and converts the branch that ends it into a jump to its closest useful postdominator, B_3. This eliminates the original loop and produces the CFG labeled "After Dead" in the margin.

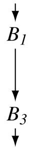


After Dead

In this form, *Clean* folds B_2 into B_1, to produce the CFG labeled "Remove B_2" in the margin. This action makes the branch at the end of B_1 redundant. *Clean* rewrites it with a jump, producing the CFG labeled "Fold the Branch" in the margin. At this point, if B_1 is B_3's sole remaining predecessor, *Clean* coalesces the two blocks into a single block.



Remove B_2



Fold the Branch

This cooperation is simpler and more effective than adding a transformation to *Clean* to handle empty loops. Such a transformation might recognize a branch from B_i to itself and, for an empty B_i, rewrite it with a jump to the branch's other target. The problem lies in determining when B_i is truly empty. If B_i contains no operations other than the branch, then the code that computes the branch condition must lie outside the loop. Thus, the transformation is safe only if the self-loop never executes. Reasoning about the number of executions of the self-loop requires knowledge about runtime values, a task that is, in general, beyond a compiler's ability. If the block contains operations, but only operations that control the branch, then the transformation would need to recognize the situation with pattern matching. In either case, this new transformation would be more complex than the four included in *Clean*. Relying on the combination of *Dead* and *Clean* achieves the desired result in a simpler, more modular fashion.

10.2.3 **Eliminating Unreachable Code**

Sometimes the CFG contains unreachable code. The compiler should find such blocks and remove them. A block can be unreachable for two distinct reasons: there may be no path through the CFG that leads to the block, or the paths that reach the block may not be executable—for example, guarded by a condition that always evaluates to false.

If the source language allows arithmetic on code pointers or labels, the compiler must preserve all blocks. Otherwise, it can limit the preserved set to blocks whose labels are referenced.

The former case is easy to handle. The compiler can perform a simple mark-sweep-style reachability analysis on the CFG. First, it initializes a mark on each block to the value "unreachable." Next, it starts with the entry and marks each CFG node that it can reach as "reachable." If all branches and jumps are unambiguous, then all unmarked blocks can be deleted. With ambiguous branches or jumps, the compiler must preserve any block that the branch or jump can reach. This analysis is simple and inexpensive. It can be done during traversals of the CFG for other purposes or during CFG construction itself.

Handling the second case is harder. It requires the compiler to reason about the values of expressions that control branches. Section 10.7.1 presents an algorithm that finds some blocks that are unreachable because the paths leading to them are not executable.

SECTION REVIEW

Code transformations often create useless or unreachable code. Many transformations simply leave the dead operations in the IR form of the code and rely on specialized transformations, such as *Dead* and *Clean*, to remove such operations. Most optimizing compilers include a set of transformations to excise dead code. Often, these passes run several times during the transformation sequence.

Dead and *Clean* do a thorough job of eliminating useless and unreachable code. However, limitations in the precision of the underlying analyses can prevent the transformations from proving that some code is dead. In particular, limits on the analysis of pointer-based values and on the precision of control-flow analysis can obscure useless or unreachable code.

REVIEW QUESTIONS

1. Experienced programmers are often certain that they do not write code that is useless or unreachable. What transformations from Chapter 8 might create useless code?

Hint: Write down the code to access `A[i,j]` where `A` is dimensioned `A[1:N,1:M]`.

2. How might the compiler, or the linker, detect and eliminate unreachable procedures? What benefits might accrue from using your technique?

10.3 CODE MOTION

Moving a computation to a point in the code where it executes less frequently than it executed in its original position should reduce the total operation count of the running program. The first transformation presented in this section, *lazy code motion*, uses code motion to speed up execution. Because loops tend to execute many more times than the code that surrounds them, much of the work in this area has focused on moving loop-invariant expressions out of loops. Lazy code motion performs loop-invariant code motion. It extends the notions originally formulated in the available expressions data-flow problem to include operations that are redundant along some, but not all, paths. It inserts code to make such expressions redundant on all paths and then removes the newly redundant expression.

Some compilers, however, optimize for other criteria. If the size of the executable code is a concern, the compiler can perform code motion to reduce the number of copies of a specific operation. The second transformation presented in this section, *hoisting*, uses code motion to eliminate duplicate instructions. It finds locations where it can insert a single operation that makes multiple copies of the same operation redundant.

10.3.1 Lazy Code Motion

Lazy code motion (LCM) uses data-flow analysis to discover both operations that are candidates for code motion and locations where it can place those operations. The algorithm operates on the IR form of the program and its CFG, rather than on SSA form. The algorithm solves three different sets of data-flow equations and derives additional sets from those results. It produces, for each edge in the CFG, a set of expressions that should be evaluated along that edge and, for each node in the CFG, a set of expressions whose upward-exposed evaluations should be removed from the corresponding block. A simple rewriting strategy interprets these sets and modifies the code.

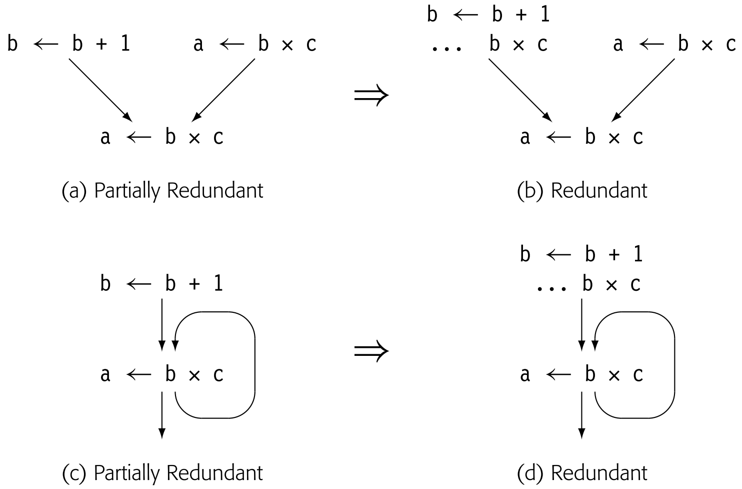
LCM operates on a CFG and IR rather than on the SSA form of the code. SSA can complicate code motion algorithms. Moving an operation that defines x_i may require renaming and, perhaps, insertion of one or more ϕ-functions.

LCM combines code motion with elimination of both redundant and partially redundant computations. Redundancy was introduced in the discussion of LVN and SVN in Chapter 8. A computation is *partially redundant* at point p if it occurs on some, but not all, paths that reach p and none of its constituent operands changes between those evaluations and p.

Redundant
An expression e is *redundant* at p if it has already been evaluated on every path that leads to p.

Partially redundant
An expression e is *partially redundant* at p if it occurs on some, but not all, paths that reach p.

Fig. 10.3 shows two ways that an expression can be partially redundant. In panel (a), the expression b × c occurs on one path leading to the join point but not on the other. To make the second computation redundant, LCM inserts an evaluation of b × c on the other path as shown in panel (b). In panel (c),



■ **FIGURE 10.3** Converting Partial Redundancies into Redundancies.

`b × c` is redundant along the loop's back edge but not along the edge that enters the loop. It is also invariant in the loop. Inserting an evaluation of `b × c` before the loop makes the evaluation inside the loop redundant, as shown in panel (d). By making the loop-invariant computation redundant and eliminating it, LCM moves it out of the loop, an optimization also called *loop-invariant code motion*.

LCM moves expression evaluations, not assignments. It will create a name, t_i, to hold `b × c` and rewrite `a ← b × c` with `a ←` t_i.

The fundamental ideas that underlie LCM were introduced in Section 9.2.4. LCM computes both available expressions and anticipable expressions. Next, LCM uses the results of these analyses to annotate each CFG edge $\langle i, j\rangle$ with a set EARLIEST(i, j) that contains the expressions for which this edge is the *earliest legal placement*. LCM then solves a third data-flow problem to find *later placements*, that is, situations where evaluating an expression after its earliest placement has the same effect. Later placements are desirable because they can shorten the lifetimes of values defined by the inserted evaluations. Finally, LCM computes its final products, two sets INSERT and DELETE, that guide its code-rewriting step.

In this context, *earliest* means the position in the CFG closest to the entry node.

Code Shape

LCM relies on several implicit assumptions about the code shape. Textually identical expressions always define the same name. Thus, each instance of $r_i + r_j$ always defines the same r_k, and the algorithm can use r_k as a proxy for $r_i + r_j$. This naming scheme simplifies the rewriting step; the optimizer can simply replace a redundant evaluation of $r_i + r_j$ with a copy from r_k,

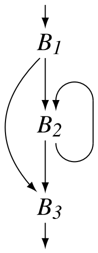
These rules are consistent with the register-naming rules from Section 4.6.1.

```
B1: loadI  1         ⇒ r10
    i2i    r10       ⇒ r4
    loadAI rarp, @n  ⇒ r11
    i2i    r11       ⇒ r5
    cmp_LE r4, r5    ⇒ r12
    cbr    r12       → B2, B3

B2: mult   r2, r3    ⇒ r15
    add    r1, r15   ⇒ r16
    i2i    r16       ⇒ r6
    addI   r4, 1     ⇒ r13
    i2i    r13       ⇒ r4
    cmp_GT r4, r5    ⇒ r14
    cbr    r14       → B3, B2

B3: ...
```

(a) A Simple Loop

Variable	a	b	c	i	n	x
VR Name	r_1	r_2	r_3	r_4	r_5	r_6

(b) Mapping from Variables to Virtual Registers



(c) Control-Flow Graph

■ **FIGURE 10.4** Example for Lazy Code Motion.

rather create a new temporary name and insert copies into r_k after each prior evaluation or $r_i + r_j$. We will add the restriction that $i < k$ and $j < k$.

LCM moves expression evaluations, not assignments. The naming discipline requires a second rule for program variables because they receive the values of different expressions. Thus, program variables are only set by register-to-register copy operations. The compiler needs a clear way to divide the name space between variables and expressions.

Finally, the compiler first assigns virtual register numbers to variables that are kept in registers, then uses higher virtual register numbers for temporary values and the computed values. Taken together, these rules ensure that a variable is always set with a copy of the form $r_i \leftarrow r_j$, where $i < j$. Thus, assignments are easy to distinguish from expression evaluations.

Combining these rules creates a clear distinction between expressions, which can be moved, and assignments, which cannot. Any operation other than a register-to-register copy operation or a control-flow transfer is an expression evaluation. Any copy operation, $r_i \Rightarrow r_j$ where $i > j$ is an assignment; if $i \leq j$, it is an expression evaluation.

Fig. 10.4(a) shows the ILOC code for the simple example loop given in the margin. The loop contains a single statement whose entire right-hand side is loop invariant. Panel (b) shows the map from variable names to register numbers; registers with subscripts larger than nine are expressions.

```
for i = 1 to n do
   x = a + b * c
```

Code for the Example

The data-flow problems used in LCM start from three distinct sets of local information. For each block *b*, LCM needs the three sets:

DEExpr(*b*):	the set of downward-exposed expressions in *b*,
UEExpr(*b*):	the set of upward-exposed expressions in *b*, and
ExprKill(*b*):	the set of expressions killed in *b*.

These sets were in Section 9.2.4. For the example program shown in Fig. 10.4, these sets have the following values:

	B_1	B_2	B_3
DEExpr	$\{r_{10}, r_{11}, r_{12}\}$	$\{r_{14}, r_{15}, r_{16}\}$	$\emptyset$
UEExpr	$\{r_{10}, r_{11}\}$	$\{r_{13}, r_{15}, r_{16}\}$	$\emptyset$
ExprKill	$\{r_{12}, r_{13}, r_{14}\}$	$\{r_{12}, r_{13}, r_{14}\}$	$\emptyset$

We show the sets for B_3 as empty, because the example abstracts away all of the code in B_3. In most realistic scenarios, B_3 would contain code and the sets for B_3 would be nonempty.

Available Expressions

To simplify the exposition, we assume that the CFG has a unique entry node n_0 and a unique exit node n_f.

The first step in LCM computes available expressions, as defined in Section 9.2.4. LCM needs information on the availability at the end of each block—AvailOut sets. An expression *e* is available on exit from block *b* if, along every path from n_0 to the end of *b*, *e* has been evaluated and none of its arguments has been subsequently defined. LCM can compute AvailOut by solving the equations:

$$\textsc{AvailIn}(n) = \bigcap_{m \in preds(n)} \textsc{AvailOut}(m)$$

$$\textsc{AvailOut}(n) = \left(\begin{array}{l} \textsc{DEExpr}(n) \cup \\ (\textsc{AvailIn}(n) \cap \overline{\textsc{ExprKill}(n)}\,) \end{array} \right)$$

AvailOut(*n*) contains any expressions that are downward exposed in *n*, plus any expressions that are available at the end of all of *n*'s CFG predecessors and not killed in *n*. The appropriate initial values are:

$$\begin{aligned} \textsc{AvailIn}(n_0) &= \emptyset \\ \textsc{AvailOut}(n) &= \{\, \textit{all expressions}\, \}, \forall\, n \neq n_0 \end{aligned}$$

The compiler can use a standard iterative solver. For the example in Fig. 10.4, this process produces the following sets:

	B_1	B_2	B_3
AVAILOUT	$\{r_{10}, r_{11}, r_{12}\}$	$\{r_{10}, r_{11}, r_{14}, r_{15}, r_{16}\}$	$\cdots$

LCM uses the AVAILOUT sets to determine possible placements for expressions in the CFG. If an expression $e \in$ AVAILOUT(b), the compiler can place an evaluation of e at the end of block b and obtain the result produced by e's most recent evaluation on any path from n_0 to b in the CFG. If $e \notin$ AVAILOUT(b), then one of e's constituent subexpressions has been modified along one of these paths, so an evaluation of e at the end of b might produce a different value. Thus, the AVAILOUT sets show the compiler how far forward in the CFG it can move the evaluation of e, ignoring any uses of e.

Anticipable Expressions

To capture information for backward motion of expressions, LCM computes anticipability. Recall, from Section 9.2.4, that an expression is anticipable at point p if and only if it is computed on every path that leaves p and produces the same value at each of those computations. Because LCM needs information about the anticipable expressions at both the start and the end of each block, we have refactored the equation to introduce a set ANTIN(n) that holds the set of anticipable expressions for the entrance of the block corresponding to node n in the CFG:

$$\text{AntOut}(n) = \bigcap_{m \in succs(n)} \text{AntIn}(m), \forall n \neq n_f$$

$$\text{AntIn}(n) = \left(\begin{array}{l} \text{UEExpr}(n) \cup \\ (\text{AntOut}(n) \cap \overline{\text{ExprKill}(n)}\,) \end{array} \right)$$

ANTIN(n) contains any expressions that are upward exposed in n, plus any expression that is anticipable at the start of each of n's CFG successors and not killed in n. The appropriate initial values are:

$$\begin{array}{lcl} \text{AntOut}(n_f) & = & \emptyset \\ \text{AntOut}(n) & = & \{\, all\ expressions\,\}, \forall n \neq n_f \end{array}$$

For the example, solving these equations yields the following sets:

	B_1	B_2	B_3
ANTIN	$\{r_{10}, r_{11}\}$	$\{r_{13}, r_{15}, r_{16}\}$	$\emptyset$
ANTOUT	$\emptyset$	$\emptyset$	$\emptyset$

AntOut provides information about the safety of hoisting an evaluation to either the start or the end of the current block. If $x \in \textsc{AntOut}(b)$, then the compiler can place an evaluation of x at the end of b, with two guarantees. First, the evaluation at the end of b will produce the same value as the next evaluation of x along any execution path in the procedure. Second, along any execution path leading out of b, the program will evaluate x before redefining any of its arguments.

Earliest Placement

Given solutions to availability and anticipability, the compiler can determine the earliest point in the program at which it can evaluate each expression. To simplify the equations, LCM assumes that evaluations are placed on a CFG edge rather than at the start or end of a specific block. Computing an edge placement lets the compiler defer the decision to insert the evaluation at the start of the edge, at the end of the edge, or in the middle of the edge (see the discussion of critical edges in Section 9.3.5, page 487).

For a CFG edge (i,j), an expression e is in $\textsc{Earliest}(i,j)$ if and only if the compiler can legally move e to (i,j), and cannot move it to any earlier edge in the CFG. The Earliest equation defines this condition:

$$\begin{aligned}\textsc{Earliest}(i,j) \;=\; & \textsc{AntIn}(\,j) \;\cap\; \overline{\textsc{AvailOut}(i)} \\ & \cap\; (\textsc{ExprKill}(i) \cup \overline{\textsc{AntOut}(i)})\end{aligned}$$

These terms define an earliest placement for e as follows:

1. $e \in \textsc{AntIn}(j)$ means that the compiler can safely move e to the head of j. Anticipability ensures that e will produce the same value as its next evaluation on any path leaving j and that each of those paths evaluates e.
2. $e \notin \textsc{AvailOut}(i)$ shows that no prior computation of e is available on exit from i. Were $e \in \textsc{AvailOut}(\,i)$, inserting e on (i,j) would be redundant.
3. The third condition encodes two cases. If $e \in \textsc{ExprKill}(i)$, the compiler cannot move e through block i because of a definition in i. If $e \notin \textsc{AntOut}(i)$, the compiler cannot move e into i because $e \notin \textsc{AntIn}(k)$ for some edge (i,k). If either is true, then e can move no further than (i,j).

LCM can ignore the third term in $\textsc{Earliest}(n_0,k)$, for any k. LCM cannot move an expression earlier than n_0. The Earliest sets for the continuing example are as follows:

	(B_1,B_2)	(B_1,B_3)	(B_2,B_2)	(B_2,B_3)
Earliest	$\{r_{13}, r_{15}, r_{16}\}$	$\emptyset$	$\{r_{13}\}$	$\emptyset$

Later Placement

The final data-flow problem in LCM determines when an earliest placement can be deferred to a later point in the CFG while achieving the same effect. Later analysis is formulated as a forward data-flow problem on the CFG with a set LATERIN(n) associated with each node and another set LATER(i,j) associated with each edge (i,j). This system of equations is defined as:

Note that LATER(i,j) is defined for edges (i,j), not for arbitrary pairs of nodes.

$$\begin{aligned}\text{LATERIN}(j) &= \bigcap_{i \in preds(j)} \text{LATER}(i,j),\ j \neq n_0 \\ \text{LATER}(i,j) &= \text{EARLIEST}(i,j)\ \cup\ (\text{LATERIN}(i) \cap \overline{\text{UEEXPR}(i)})\end{aligned}$$

The appropriate initial values are:

$$\begin{aligned}\text{LATERIN}(n_0) &= \emptyset \\ \text{LATERIN}(n) &= \{\,\textit{all expressions}\,\},\ \forall n \neq n_0\end{aligned}$$

The compiler can use a standard iterative solver on these equations. These equations have a unique fixed point.

An expression $e \in$ LATERIN(k) if and only if every path that reaches k includes an edge (p,q) such that $e \in$ EARLIEST(p,q), and the path from q to k neither redefines e's operands nor contains an evaluation of e that an earlier placement of e would anticipate. The EARLIEST term in the equation for LATER ensures that LATER(i,j) includes EARLIEST(i,j). The rest of that equation puts e into LATER(i,j) if e can be moved forward from i ($e \in$ LATERIN(i)) and a placement at the entry to i does not anticipate a use in i ($e \notin$ UEEXPR(i)).

Given LATER and LATERIN sets, $e \in$ LATERIN(i) implies that the compiler can move the evaluation of e forward through i without losing any benefit—that is, there is no evaluation of e in i that an earlier evaluation would anticipate, and $e \in$ LATER(i,j) implies that the compiler can move an evaluation of e in i into j.

For the example, these equations produce the following sets:

	B_1	B_2	B_3
LATERIN	∅	{r13}	∅

	(B_1,B_2)	(B_1,B_3)	(B_2,B_2)	(B_2,B_3)
LATER	{r13,r15,r16}	∅	{r13}	∅

Rewriting the Code

The final step in performing LCM is to rewrite the code so that it capitalizes on the knowledge derived from the data-flow computations. To drive the rewriting process, LCM computes two additional sets, INSERT and DELETE.

The INSERT set specifies, for each edge, the computations that LCM should insert on that edge.

$$\text{INSERT}(i, j) = \text{LATER}(i, j) \cap \overline{\text{LATERIN}(j)}$$

If i has only one successor, LCM can insert the computations at the end of i. If j has only one predecessor, it can insert the computations at the entry of j. If neither condition applies, the edge (i,j) is a critical edge and the compiler should split it by inserting a block in the middle of the edge to evaluate the expressions in INSERT(i,j).

The DELETE set specifies, for a block, which computations LCM should delete from the block.

$$\text{DELETE}(i) = \text{UEEXPR}(i) \cap \overline{\text{LATERIN}(i)}, \quad i \neq n_0$$

DELETE(n_0) is $\emptyset$ because n_0 has no predecessor. If $e \in$ DELETE(i), then the first computation of e in i is redundant after all the insertions have been made. Any subsequent evaluation of e in i that has upward-exposed uses—that is, the operands are not defined between the start of i and the evaluation—can also be deleted. Because all evaluations of e define the same name, the compiler need not rewrite subsequent references to the deleted evaluation. Those references will naturally refer to earlier evaluations of e that will produce the same result.

For the example, the INSERT and DELETE sets are simple:

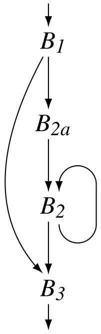
	(B_1,B_2)	(B_1,B_3)	(B_2,B_2)	(B_2,B_3)
INSERT	$\{r_{15}, r_{16}\}$	$\emptyset$	$\emptyset$	$\emptyset$

	B_1	B_2	B_3
DELETE	$\emptyset$	$\{r_{15}, r_{16}\}$	$\emptyset$

The compiler interprets the INSERT and DELETE sets and rewrites the code as shown in Fig. 10.5(a). LCM deletes the expressions that define r_{15} and r_{16} from B_2 and inserts them on the edge (B_1,B_2).

```
B1:  loadI   1         ⇒ r10
     i2i     r10       ⇒ r4
     loadAI  rarp, @n  ⇒ r11
     i2i     r11       ⇒ r5
     cmp_LE  r4, r5    ⇒ r12
     cbr     r12       → B2a, B3
B2a: mult    r2, r3    ⇒ r15
     add     r1, r15   ⇒ r16
     jump              → B2
B2:  i2i     r16       ⇒ r6
     addI    r4, 1     ⇒ r13
     i2i     r13       ⇒ r4
     cmp_GT  r4, r5    ⇒ r14
     cbr     r14       → B3,B2
B3:  ...
```

(a) The Transformed Code



(b) Its CFG

■ **FIGURE 10.5** Example After Lazy Code Motion.

Since B_1 has two successors and B_2 has two predecessors, (B_1,B_2) is a critical edge. Thus, LCM splits the edge, creating a new block B_{2a} to hold the inserted computations of r_{15} and r_{16}. Splitting (B_1,B_2) adds an extra `jump` to the code. Code generation and block placement should strive to place B_{2a} so that the end-of-block jump becomes a fall-through case.

Notice that LCM leaves the copy defining r_6 in B_2. (Recall that r_6 holds the variable x.) LCM moves expressions, not assignments. If the copy is not needed, subsequent copy coalescing, either in the register allocator or as a standalone pass, should discover that fact and eliminate the copy operation.

10.3.2 **Code Hoisting**

Code motion techniques can also be used to reduce the size of the compiled code. *Code hoisting* provides one direct way to accomplish this goal. It uses the results of anticipability analysis in a particularly simple way.

If an expression $e \in$ ANTOUT(b), for some block b, that means that e is evaluated along every path that leaves b and that an evaluation of e at the end of b would make the first evaluation along each path redundant. To reduce code size, the compiler can insert an evaluation of e at the end of b and replace the first occurrence of e on each path that leaves b with a reference to the previously computed value. The effect of this transformation is to replace multiple copies of the evaluation of e with a single copy, reducing the overall number of operations in the compiled code.

The equations for ANTOUT ensure that none of e's operands is redefined between the end of b and the next evaluation along each path that leaves b.

Some authors use the name VERYBUSY for ANTOUT when used in hoisting.

To replace those expressions directly, the compiler would need to locate them. It could insert *e*, then solve another data-flow problem, proving that the path from *b* to some evaluation of *e* is clear of definitions for *e*'s operands. Alternatively, it could traverse each of the paths leaving *b* to find the first block where *e* is defined—by looking in the block's UEEXPR set. Each of these approaches seems complicated.

A simpler approach has the compiler visit each block *b* and insert an evaluation of *e* at the end of *b*, for every expression $e \in$ ANTOUT(b). If the compiler uses a uniform discipline for naming, as suggested in the discussion of LCM, then each evaluation will define the appropriate name. Either LCM or SVN can then remove the newly redundant expressions.

SECTION REVIEW

Compilers perform code motion for two primary reasons. Moving an operation to a point where it executes fewer times than it would in its original position should reduce execution time. Moving an operation to a point where one instance replaces several should reduce code size. This section presented an example of each.

LCM is a classic example of a data-flow driven global optimization. It identifies redundant and partially redundant expressions, computes the best place for them, and moves them. By definition, any loop-invariant expression is either redundant or partially redundant; LCM moves a large class of loop invariant expressions out of loops. Hoisting takes a much simpler approach; it finds operations that are redundant on every path that leaves some point p and replaces the redundant occurrences with one evaluation at p. Thus, hoisting is usually done to reduce code size.

One common form of sinking is called *cross jumping*.

REVIEW QUESTIONS

1. Hoisting discovers the situation in which some expression e exists along each path that leaves point p and each of those occurrences can be replaced safely with an evaluation of e at p. Formulate the symmetric and equivalent optimization, *code sinking*, that discovers when multiple expression evaluations can safely be moved forward in the code—from points that precede p to p.
2. Consider what will happen if you apply your code-sinking transformation at link time, when all the code for the entire application is present. What opportunities might it find in the procedure linkage code?

10.4 SPECIALIZATION

In most compilers, the front end determines the shape of the IR program before it performs any detailed analysis of the code. Of necessity, this produces general code that works in any context that the running program might encounter. With analysis, however, the compiler can often learn enough to narrow the contexts in which the code must operate. The compiler can then specialize the sequence of operations in ways that capitalize on its knowledge of the context in which that code will execute.

Important techniques that perform specialization appear in other sections of this book. Constant propagation, described in Sections 9.3.6 and 10.7.1, analyzes a procedure to find variables and expressions that have known constant values; it then folds those values directly into the computation. Interprocedural constant propagation, introduced in Section 9.4.2, applies the same ideas at the whole-program scope. Operator strength reduction, presented in Section 10.7.2, replaces inductive sequences of expensive computations with equivalent sequences of faster operations. Peephole optimization, introduced in Section 11.3, uses pattern matching over short instruction sequences to find local improvement. Value numbering, shown in Sections 8.4.1 and 8.5.1, systematically simplifies the IR form of the code by applying algebraic identities and local constant folding. Each of these techniques implements a form of specialization.

Optimizing compilers rely on these general techniques to improve code. In addition, most optimizing compilers contain specialization techniques that specifically target properties of the source languages or applications that the compiler writer expects to encounter. The rest of this section presents three such techniques that target specific inefficiencies at procedure calls: tail-call optimization, leaf-call optimization, and parameter promotion.

10.4.1 Tail-Call Optimization

When the last action that a procedure takes is a call, we refer to that call as a tail call. The compiler can specialize tail calls to their contexts in ways that eliminate much of the overhead from the procedure linkage. To understand how the opportunity for improvement arises, consider what happens when o calls p and p calls q. When q returns, it executes its epilog sequence and jumps back to p's postreturn sequence. Execution continues in p until p returns, at which point p executes its epilog sequence and jumps to o's postreturn sequence.

If the call from p to q is a tail call, then no useful computation occurs between the postreturn sequence and the epilog sequence in p. Thus, any code that preserves and restores *p*'s state, beyond what is needed for the return from *p* to *o*, is useless. A standard linkage, as described in Section 6.5, spends much of its effort to preserve state that is useless in the context of a tail call.

At the call from *p* to *q*, the minimal precall sequence must evaluate the actual parameters at the call from *p* to *q* and adjust the access links or the display if necessary. It need not preserve any caller-saves registers, because they cannot be live. It need not allocate a new AR, because *q* can use *p*'s AR, provided that *p*'s local data area is large enough for *q*. (If not, the compiler can arrange for *p*'s prolog to allocate a larger data area.) It must leave intact the context created for a return to *o*, namely the return address and caller's ARP that *o* passed to *p*, and any callee-saves registers that *p* preserved in its own AR. Finally, the precall sequence must jump to a tailored prolog sequence for *q*.

That context will cause the epilog code for *q* to return control directly to *o*.

In this scheme, *q* needs a custom prolog sequence to match the custom precall sequence in *p*. It only needs to save the parts of *p*'s state that allow a return to *o*. *q*'s prolog does not preserve callee-saves registers, for two reasons. First, values computed by *p* into those registers are dead. Second, the callee-saves values that *p* saved must be restored on return to *o* from *q*. Thus, *q*'s custom prolog should simply initialize local variables and values that *q* needs and branch into the code for *q*.

With these changes to *p*'s precall sequence and the *q*'s prolog, the tail call avoids saving and restoring *p*'s state, which eliminates much of the overhead of the call. Of course, once *p*'s precall has been tailored in this way, *p*'s sequences are unreachable. Standard techniques such as *Dead* and *Clean* will not discover that fact, because they assume that the interprocedural jumps to their labels are executable. As the optimizer tailors the call, it can eliminate these dead sequences.

With a little care, the optimizer can arrange for the operations in the tailored prolog for *q* to appear as the last operations in its more general prolog. In this scheme, the tail call from *p* to *q* simply jumps to a point farther into the prolog sequence than would a normal call from some other routine.

If the tail call is a self-recursive call—that is, *p* and *q* are the same procedure—then tail-call optimization can produce particularly efficient code. In a self-recursive tail call, the entire precall sequence devolves to argument evaluation and a branch back to the top of the routine. An eventual return out of the recursion requires one branch, rather than one branch per recursive invocation. The resulting code can rival a traditional loop for efficiency.

10.4.2 **Leaf-Call Optimization**

Some of the overhead involved in a procedure call arises from the need to prepare for calls that the callee might make. A procedure that makes no calls, called a leaf procedure, creates opportunities for specialization. The compiler can easily identify a leaf procedure; it makes no calls.

During translation of a leaf procedure, the compiler can avoid inserting operations whose sole purpose is to set up for subsequent calls. For example, the procedure prolog code may save the return address from a register into a slot in the AR. That action is unnecessary unless the procedure itself makes another call. If the register that holds the return address is needed for some other purpose, the register allocator can spill the value. Similarly, if the implementation uses a display to provide addressability for nonlocal variables, as described in Section 6.4.3, a leaf procedure can avoid the display update in the prolog sequence.

The other reason to store the return address is to allow later tools to unwind the call stack. When use of such tools is expected, the compiler should leave the save operation intact.

The register allocator should use caller-saves registers before callee-saves registers in a leaf procedure. To the extent that it can avoid the callee-saves registers, it can eliminate saves and restores in the prolog and epilog. In small leaf procedures, the compiler may be able to avoid all use of callee-saves registers. If the compiler has access to both the caller and the callee, it can do better; for leaf procedures that need fewer registers than the caller-save set includes, it can avoid some of the register saves and restores in the caller as well.

In addition, the compiler can avoid the runtime overhead of activation-record allocation for leaf procedures. In an implementation that heap allocates ARs, that cost can be significant. In an application with a single thread of control, the compiler can allocate statically the AR of any leaf procedure. A more aggressive compiler might allocate one static AR that is large enough to work for any leaf procedure and have all the leaf procedures share that AR.

If the compiler has access to both the leaf procedure and its callers, it can allocate space for the leaf procedure's AR in each of its callers' ARs. This scheme amortizes the AR allocation cost over at least two calls—the invocations of the caller and the leaf procedure. If the caller invokes the leaf procedure multiple times, the savings are multiplied.

10.4.3 **Parameter Promotion**

Ambiguous memory references prevent the compiler from keeping values in registers. Sometimes, the compiler can employ the results of special case analysis, analysis of pointer values, or analysis of array subscript

Promotion
a category of transformations that move an ambiguous value into a local scalar name to expose it to register allocation

expressions to prove that a potentially ambiguous value refers, in reality, to just one memory location. In such cases, the compiler can rewrite the code to move that value into a scalar local variable, where the register allocator can keep it in a register. This kind of transformation is often called *promotion*. The analysis to promote array references or pointer-based references is beyond the scope of this book. However, a simpler case can illustrate these transformations equally well.

Call-by-reference parameters can be ambiguous. The call site may pass the same variable in two or more parameter slots. It might pass a global variable as an actual parameter. Unless the compiler performs interprocedural analysis to rule out such possibilities, it must treat every call-by-reference formal parameter as ambiguous inside the callee. The potential ambiguity forces the compiler to keep the parameter's value in memory rather than in a register.

The compiler can create a new name to hold the promoted value: a local scalar value in a near-source IR or a virtual register in a lower-level IR.

If the compiler can prove that the actual parameter must be unambiguous in the callee, it can promote the parameter's value. It can copy the value into its temporary name at the start of the procedure and copy it back before the procedure returns. The temporary name can then compete for registers with the other unambiguous scalar values. If the callee cannot modify the parameter, the compiler could convert the parameter to use call-by-value.

To apply this transformation to a procedure *p*, the optimizer must identify all of the call sites that can invoke *p*. It can either prove that the transformation applies at all of those call sites or it can clone *p* to create a copy that handles the promoted values (see Section 10.6.2).

SECTION REVIEW

Specialization includes many techniques that tailor general-purpose computations to their detailed contexts. Other chapters and sections present powerful global and regional specialization techniques, such as constant propagation, peephole optimization, and operator strength reduction.

This section focused on optimizations that the compiler can apply to the code that implements procedure calls. Tail-call optimization is a valuable tool that converts tail recursion to a form that rivals conventional iteration for efficiency; it applies to nonrecursive tail calls as well. Leaf procedures offer special opportunities for improvement because the callee can omit major portions of the standard linkage sequence. Parameter promotion is one example of a class of important transformations that remove inefficiencies related to ambiguous references.

REVIEW QUESTIONS

1. Parameter promotion transforms the code to move an unambiguous call-by-reference parameter in a local scalar variable. An analogous transformation could look at a loop and move a pointer-based value (e.g., `*p` in C) into a local scalar variable throughout the loop. What safety conditions would the compiler need to prove to perform such scalar promotion on pointer-based values?
2. Inline substitution might be an alternative to the procedure-call optimizations in this section. How might you apply inline substitution in each case? How might the compiler choose the more profitable alternative?

10.5 REDUNDANCY ELIMINATION

A computation $x+y$ is redundant at some point p in the code if, along every path that reaches p, $x+y$ has already been evaluated and x and y have not been modified since the evaluation. Redundant computations typically arise as artifacts of translation or optimization.

We have already presented three effective techniques for redundancy elimination: local value numbering (LVN) in Section 8.4.1, superlocal value numbering (SVN) in Section 8.5.1, and lazy code motion (LCM) in Section 10.3.1. These algorithms cover the span from simple and fast (LVN) to complex and comprehensive (LCM). While all three methods differ in scope, the primary distinction between them lies in the way that they establish that two values are identical. Section 10.5.1 explores this issue in detail. Section 10.5.2 presents one more version of value numbering, a dominator-based technique.

10.5.1 Value Identity Versus Name Identity

LVN introduced a simple mechanism to prove that two expressions had the same value. LVN relies on two principles. It assigns each value a unique identifying number—its value number. It assumes that two expressions produce the same value if they have the same operator and their operands have the same value numbers. These simple rules allow LVN to find a broad class of redundant operations—any operation that produces a preexisting value number is redundant.

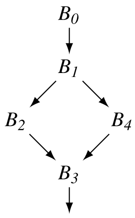
With these rules, LVN can prove that $2+a$ has the same value as $a+2$ or as $2+b$ when a and b have the same value number. It cannot prove that $a+a$ and $2 \times a$ have the same value because they have different operators.

Similarly, it cannot prove the $a+0$ and a have the same value. Thus, we extended LVN with algebraic identities to handle many well-defined cases not covered by the original rule. The table in Fig. 8.3 on page 398 shows the range of identities that LVN can handle. SVN extends the optimization to larger scopes.

By contrast, LCM relies on names to prove that two values have the same number. If LCM sees $a+b$ and $a+c$, it assumes that they have different values because b and c have different names. It relies on a lexical comparison—name identity. The underlying data-flow analyses cannot directly accommodate the notion of value identity; data-flow problems operate over a predefined name space and propagate facts about those names over the CFG. The kind of ad-hoc comparisons used in LVN do not fit into the data-flow framework.

The compiler can improve the effectiveness of LCM by encoding value identity into the code's name space before applying LCM. The compiler first applies a value numbering technique, such as DVNT from the next subsection, to find value numbers. Next, it rewrites the code in terms of the value numbers. Finally, it applies LCM to the transformed code. The resulting code should reflect the strengths of both approaches. It can find improvements that neither technique could find on its own.

10.5.2 Dominator-Based Value Numbering



CFG Fragment

Chapter 8 described two value numbering algorithms, LVN and SVN. While SVN discovers more redundancies than LVN, it still misses some opportunities because it only propagates information along paths through extended basic blocks (EBBs). For example, in the CFG fragment shown in the margin, SVN will process the paths $\langle B_0, B_1, B_2\rangle$ and $\langle B_0, B_1, B_4\rangle$. Thus, it optimizes both B_2 and B_4 in the context of the prefix path $\langle B_0, B_1\rangle$. Because B_3 forms its own degenerate EBB, SVN optimizes B_3 without any prior context. SVN cannot propagate information across the join point and into B_3.

From an algorithmic point of view, SVN begins each block with a table that includes the results of all predecessors on its EBB path. Block B_3 forms a degenerate EBB with no predecessors, so it has no prior context. To improve on that situation, we must answer the question: on what state could B_3 rely?

B_3 cannot rely on values computed in either B_2 or B_4, since neither lies on every path that reaches B_3. By contrast, B_3 can rely on values computed in B_0 and B_1, since they occur on every path that reaches B_3. Thus, we might extend value numbering for B_3 with information about computations in B_0 and B_1. We must, however, account for the impact of assignments in the intervening blocks, B_2 or B_4.

Consider an expression, $x+y$, that occurs at the end of B_1 and again at the start of B_3. If neither B_2 or B_4 redefines x or y, then the evaluation of $x+y$ in B_3 is redundant and the optimizer can reuse the value computed in B_1. On the other hand, if either of those blocks redefines x or y, then the evaluation of $x+y$ in B_3 computes a distinct value from the evaluation in B_1 and the evaluation is not redundant. We need an algorithmic way to ensure that the algorithm can distinguish between these two cases.

Fortunately, the SSA name space encodes precisely this distinction. In SSA, a name that is used in some block B_i can only enter B_i in one of two ways. Either the name is defined by a ϕ-function at the top of B_i, or it is defined in some block that dominates B_i. Thus, an assignment to x in either B_2 or B_4 creates a new name for x and forces the insertion of a ϕ-function for x at the head of B_3. That ϕ-function creates a new SSA name for x and the renaming process changes the SSA name used in the subsequent evaluation of $x+y$ in B_3. Thus, SSA form encodes the presence or absence of an intervening assignment in B_2 or B_4 directly into the name space. A value-numbering algorithm that uses SSA names can avoid the complications posed by those intervening assignments.

This discussion has a subtle consequence. Data-flow problems that use KILL sets, such as VARKILL or EXPRKILL, can be reformulated over SSA names to eliminate those sets.

The resulting framework can be noticeably cheaper to solve.

The other major question that we must answer before we can extend SVN is: given a block such as B_3, how do we locate the most recent predecessor with information that the algorithm can use? The IDOM relation, discussed in Section 9.2.1, captures precisely this effect. B_1 is IDOM(B_3), so it is the closest node that occurs on every path from the entry node to B_3. For a block b, the algorithm can use the tables from IDOM(b) as an initial state.

The dominator-based value numbering technique (DVNT) builds on the ideas in SVN. It uses a sheaf of tables to map expressions to value numbers. It creates a new table for each block and discards that table when it is no longer needed. To simplify bookkeeping, DVNT simply uses the SSA names as value numbers. Thus, if $t_k \leftarrow a_i \times b_j$ is the first evaluation of $a_i \times b_j$, then the value number for $a_i \times b_j$ is t_k.

The SSA name refers uniquely to the result of one definition point.

Fig. 10.6 shows the algorithm. It takes the form of a recursive procedure that the optimizer invokes on a procedure's entry block. It follows both the CFG for the procedure, represented by the dominator tree, and the flow of values in the SSA form. For each block B, DVNT takes three steps: it processes any ϕ-functions that exist in B; it value numbers the assignments; and it propagates information into B's successors and recurs on B's children in the dominator tree.

```
DVNT(B)
    allocate a new table for B and link it into the sheaf
    for each $\phi$-function p of the form "n ← $\phi(\ldots)$" in B do
        if p is meaningless then
            VN[n] ← the value number for p's first argument
            delete p from the code
        if p is redundant with some $\phi$-function q in B then
            VN[p] ← VN[q]
            delete p from the code
        else
            VN[n] ← n
            Add p to the hash table with value number n
    for each assignment "x ← y op z" in B, in order, do
        overwrite y with VN[y]
        overwrite z with VN[z]
        let expr ← "y op z"
        if expr can be simplified to expr′ then
            rewrite the assignment as "x ← expr′ "
            expr ← expr′
        if expr has a value number v in the hash table then
            VN[x] ← v
            delete the assignment from the code
        else
            VN[x] ← x
            add expr to the hash table with value number x
    for each successor s of B do
        adjust the $\phi$-function inputs in s
    for each child c of B in the dominator tree
        DVNT(c)
    free the scope for B
```

■ **FIGURE 10.6** Dominator-Based Value Numbering Technique.

Process the ϕ-Functions in B

DVNT must assign each ϕ-function p a value number. If p is meaningless—that is, all its arguments have the same value number—DVNT sets its value number to the value number for one of its arguments and deletes p. If p is redundant—that is, it produces the same value number as another ϕ-function q in B—DVNT assigns p the value number of q. DVNT then deletes p.

Otherwise, the ϕ-function computes a new value. DVNT assigns it a value number—the SSA number of its target. If the code reaches this point, then either the ϕ-function has a unique combination of arguments, or some of its arguments do not yet have a value number. In either case, DVNT cannot simplify the ϕ-function.

Process the Assignments in B

DVNT iterates over the assignments in *B* and processes them in a manner analogous to LVN and SVN. One subtlety arises from the use of SSA names as value numbers. When the algorithm encounters a statement *x* ← *y op z*, it can simply replace *y* with *VN[y]* because the name in *VN[y]* holds the same value as *y*.

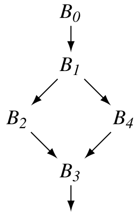
Recall that the algorithm to build SSA does not allow uninitialized names.

Propagate Information to B's Successors

Once DVNT has processed all the ϕ-functions and assignments in *B*, it visits each of *B*'s CFG successors *s* and updates ϕ function arguments that correspond to values flowing across the edge (*B*,*s*). It records the current value number for the argument in the ϕ-function by overwriting the argument's SSA name. Next, the algorithm recurs on *B*'s children in the dominator tree. Finally, it frees the hash table that it used for *B*.

Notice the similarity between this step and the corresponding step in the renaming phase of the SSA construction.

This recursive scheme takes DVNT on a preorder walk of the dominator tree, which ensures that the appropriate tables have been constructed before it visits a block. This order can produce a counterintuitive traversal; for the CFG in the margin, the algorithm could visit B_3 before either B_2 or B_4. Since the only facts that the algorithm can use in B_3 are those discovered processing B_0 and B_1, the relative ordering of B_2, B_4, and B_3 is not only unspecified, it is also irrelevant.



SECTION REVIEW

Redundancy elimination assumes that it is faster to reuse a value than to recompute it. LCM and DVNT find redundant computations and eliminate duplicate evaluations. LCM uses name-based identity while DVNT uses value-based identity. These different notions of identity can find different redundancies.

Both LCM and DVNT eliminate redundant computations. LCM removes the *evaluation* of redundant and partially redundant expressions; it does not eliminate assignments. Value numbering also removes assignments; it does not find partial redundancies. To combine the strengths of both techniques, some compilers use value-numbering to encode value identity into the name space and a name-based technique such as LCM to rewrite the code.

REVIEW QUESTIONS

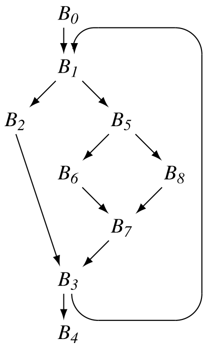
1. DVNT does not propagate a value along a loop-closing edge—a back edge in the call graph. LCM will propagate information along such edges. Write several examples of redundant expressions that a true "global" technique such as LCM can find that DVNT cannot.
2. The DVNT algorithm resembles the renaming phase of the SSA construction algorithm. Can you reformulate the renaming phase so that it performs value numbering as it renames values? What impact would this change have on the size of the SSA form for a procedure?

10.6 ENABLING OTHER TRANSFORMATIONS

Often, an optimizer includes passes whose primary purpose is to create or expose opportunities for other transformations. In some cases, a transformation changes the shape of the code to make it more amenable to optimization. In other cases, the transformation creates a point in the code where specific conditions hold that make another transformation safe or profitable. By directly creating the necessary code shape, these enabling transformations reduce the sensitivity of the optimizer to the shape of the source program.

Several enabling transformations are described in other parts of the book. Both loop unrolling (Section 8.5.2) and inline substitution (Section 8.7.1) obtain most of their benefits by creating context for other optimizations. (In each case, the transformation eliminates some overhead, but the larger effect comes from subsequent application of other optimizations.) The tree-height balancing algorithm (Section 8.4.2) does not eliminate any operations, but it creates a code shape that can produce better results from instruction scheduling. This section presents four enabling transformations: *superblock cloning*, *procedure cloning*, *loop unswitching*, and *renaming*.

10.6.1 Superblock Cloning



CFG for Superblock Cloning

Often, the optimizer's ability to transform the code is limited by path-specific information in the code. Imagine using SVN on the CFG shown in the margin. The fact that blocks B_3 and B_7 have multiple predecessors may limit the optimizer's ability to improve those blocks. If, for example, B_6 assigned x the value 7 and B_8 assigned x the value 13, a use of x in B_7 would appear to receive the value $\perp$, even though the value is known and predictable along each path leading to B_7.

In such circumstances, the compiler can clone blocks to create code that is better suited for the transformation. In this case, it might create two copies of B_7, say B_{7a} and B_{7b}, and redirect the incoming edges as $\langle B_6, B_{7a}\rangle$ and $\langle B_8, B_{7b}\rangle$. With this change, the optimizer could propagate the value 7 for x into B_{7a} and the value 13 for x into B_{7b}.

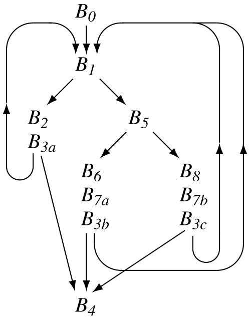
As an additional benefit, since B_{7a} and B_{7b} both have unique predecessors, the compiler can merge the blocks to create one block from B_6 and B_{7a} and another from B_8 and B_{7b}. This change eliminates the block-ending jump in B_6 and B_8 and, potentially, allows for further improvement in optimization.

Backward branch
a CFG edge (i, j) where the depth-first number of i is less than or equal to the depth-first number of j

Loop-closing branch
a CFG edge (i, j) where $j \in \text{DOM}(i)$.

An issue in this kind of cloning is, when should the compiler stop cloning? One technique, called *superblock cloning*, is widely used to create additional context for instruction scheduling inside loops. In superblock cloning, the optimizer starts with a loop head—the entry to a loop—and clones each path until it reaches a backward branch.

Applying this technique to the example CFG produces the modified CFG shown in the margin. B_1 is the loop header. Each of the nodes in the loop body has a unique predecessor. If the compiler applies a superlocal optimization (one based on EBBs), every path that it finds will encompass a single iteration of the loop body. (To find longer paths, the optimizer could need to unroll the loop before superblock cloning.)



CFG After Superblock Cloning

Superblock cloning enables other optimizations in three principal ways:

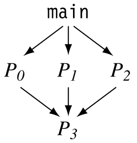
1. *It creates longer blocks.* Longer blocks let local and superlocal optimization see more context. While LVN extends nicely from one block to larger regions, other optimizations do not. In the transformed CFG, local techniques can be applied directly to the merged blocks. For example, the compiler could apply the tree-height balancing algorithm to $\langle B_6, B_{7a}, B_{3b}\rangle$, rather than balancing B_6, B_{7a}, and B_{3b}, by themselves.
2. *It eliminates branches.* Combining two blocks eliminates a branch between them. Branches take time to execute. They also can disrupt performance-critical mechanisms in the processor, such as instruction prefetch. The net effect is to reduce the number of operations that execute and to make hardware prediction mechanisms more effective.
3. *It creates points where optimization can occur.* When cloning eliminates a join point in the CFG, it creates new points in the program where the compiler can derive more precise knowledge about the runtime context. The transformed code may present opportunities for optimization that exist nowhere in the original code.

Of course, cloning has costs, too. It duplicates individual operations, which causes code growth. The larger code may run more quickly because it avoids some end-of-block jumps. It may run more slowly if its size causes more instruction cache misses. In applications where the user cares more about code space than runtime speed, superblock cloning may be counterproductive.

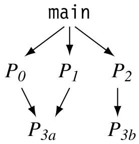
10.6.2 Procedure Cloning

Inline substitution, described in Section 8.7.1, has effects similar to superblock cloning. For a call from *p* to *q*, it creates a unique copy of *q* and merges it with the call site in *p*. It achieves the same effects as superblock cloning, including specialization to a particular context, elimination of some control-flow operations, and increased code size.

In some cases, cloning a procedure can achieve some of the benefits of inline substitution with less code growth. The key observation is that procedure cloning need not create a separate clone for each call site. Rather, it should create a clone for each interesting optimization context. The compiler creates multiple copies of the callee and assigns calls with similar context to the same clone.



Original Call Graph



After Cloning P_3

Consider, for example, the simple call graph in the margin. Assume that P_3's behavior depends strongly on one of its input parameters; for a value of one, the compiler can generate code that provides efficient memory access, while for other values, it produces much larger, slower code. Further, assume that P_0 and P_1 both pass the value 1 to P_3, while P_2 passes it the value 17.

Constant propagation across the call graph does not help here because it must compute P_3's parameter as $1 \wedge 1 \wedge 17 = \bot$, which does not allow specialization in P_3. Procedure cloning can create a place where the parameter is always 1, as with P_{3a} in the graph in the margin. The call that inhibits optimization, (P_2,P_3) in the original call graph, is assigned to P_{3b}. The compiler can generate optimized code for P_{3a} and generic code for P_{3b}.

10.6.3 Loop Unswitching

Loop unswitching hoists loop-invariant control-flow operations out of a loop. If the predicate in an `if–then–else` construct is loop invariant, then the compiler can rewrite the loop to pull the `if–then–else` out of the loop and generate a tailored copy of the loop inside each half of the new `if–then–else`. Fig. 10.7 shows a brief example.

Unswitching is an enabling transformation; it allows the compiler to tailor loop bodies in ways that are otherwise hard to achieve. After unswitching, the remaining loops contain less control flow. They execute fewer branches and other operations to support those branches. This can lead to better scheduling, better register allocation, and faster execution. If the original

```
do i = 1 to n
    if (x > y) then
             a(i) = b(i) * x
        else a(i) = b(i) * y
```

(a) Original Loop

```
if (x > y) then
    do i = 1 to n
        a(i) = b(i) * x
else
    do i = 1 to n
        a(i) = b(i) * y
```

(b) Unswitched Version

■ **FIGURE 10.7** Unswitching a Short Loop.

loop contained loop-invariant code that was inside the `if–then–else`, then LCM could not move it out of the loop. After unswitching, LCM easily finds and removes such redundancies.

Unswitching also has a simple, direct effect that can improve a program: it moves the branch from the loop-invariant conditional out of the loop. Moving control flow out of loops is difficult. Techniques based on data-flow analysis, like LCM, have trouble moving such constructs because the transformation modifies the CFG on which the analysis relies. Techniques such as SVN and DVNT can recognize cases where the predicates controlling `if–then–else` constructs are constant or redundant but can neither simplify the control flow nor move the construct out of a loop.

10.6.4 **Renaming**

Most scalar transformations rewrite or reorder operations in the code. We have seen, in several contexts, that the choice of names can either obscure or expose opportunities for improvement. For example, in LVN, converting the names in a block to SSA names exposed some opportunities for reuse that would otherwise be difficult to capture.

For many transformations, careful construction of the "right" name space can expose additional opportunities, either by making more facts visible to analysis or by avoiding some of the side effects that arise from reuse of storage. As discussed in Section 10.5.1, the compiler writer can improve the effectiveness of LCM by encoding value identity into the name space, where LCM's name-based, data-flow approach will discover the redundancy and perform the appropriate redundancy elimination and code motion. Careful renaming exposes more opportunities to LCM and makes it more effective.

In a similar way, names matter to instruction scheduling. In a scheduler, names encode the data dependences that constrain the placement of operations in the scheduled code. When the reuse of a name reflects the actual flow of values, that reuse provides critical information required for

The illusion of a constraint introduced by naming is often called *false sharing*.

correctness. If reuse of a name occurs because a prior pass has compressed the name space, then the reuse may unnecessarily constrain the schedule. For example, the register allocator places distinct values into the same physical register to improve register utilization. If the compiler performs allocation before scheduling, the allocator can introduce apparent constraints on the scheduler that are not strictly required by the original code.

Renaming is a subtle issue. Individual transformations can benefit from name spaces with different properties. Compiler writers have long recognized that moving and rewriting operations can improve programs. In the same way, they should recognize that renaming can improve optimizer effectiveness. As SSA has shown, the compiler need not be bound by the name space introduced by the programmer or by the compiler's front end. Renaming is a fertile ground for future work

SECTION REVIEW

As we saw in Chapter 7, the shape of the IR affects the code that the compiler can generate. The techniques in this section rewrite the code to create opportunities for other optimizations. They use replication, selective rewriting, and renaming to create places in the code that can be improved by specific transformations.

Cloning, at either the block or procedure level, achieves its effects by eliminating control-flow merge points. It can simplify paths and combine blocks or procedures. Loop unswitching moves control structures out of a loop to make them both more amenable to optimization. Renaming is a powerful idea with widespread application, including redundancy elimination, strength reduction, scheduling, and register allocation.

REVIEW QUESTIONS

1. Superblock cloning creates new opportunities for other optimizations. Consider tree-height balancing. How much can superblock cloning help? Can you envision a transformation to follow superblock cloning that would expose more opportunities for tree-height balancing? For SVN, how might the results of using SVN after cloning compare to the results of running LCM on the same code?
2. Procedure cloning attacks some of the same inefficiencies as inline substitution. Is there a role for both of these transformations in a single compiler? What are the potential benefits and risks of each transformation? How might a compiler choose between them?

THE SSA GRAPH

In some algorithms, viewing the SSA form of the code as a graph simplifies either the discussion or the implementation. Both sparse conditional constant propagation (SSCP) and the strength reduction algorithm operate on SSA form viewed as a graph.

In SSA form, each name corresponds to exactly one definition. A use of that name in operation i can be interpreted as a chain from i back to the name's definition. The compiler can easily construct maps from uses to their definitions and definitions to their uses during the SSA construction. These maps can be interpreted as a graph, with edges that connect definitions to uses and reflect the flow of values in the original code.

We draw SSA graphs with edges that run from a use to its corresponding definition, which indicates the relationship implied by the SSA names. The compiler may need to traverse the edges in both directions. SCCP propagates values from definitions to uses. OSR moves, primarily, from uses to definitions. The compiler writer can easily build the graph to allow traversal in either direction.

10.7 ADVANCED TOPICS

Most of the examples in this chapter have been chosen to illustrate a specific effect that the compiler can use to speed up the executable code. Sometimes, performing two optimizations together can produce results that cannot be obtained with any combination of applying them separately. The next subsection shows one such example: combining constant propagation with unreachable code elimination. Section 10.7.2 presents a second, more complex example of specialization: operator strength reduction with linear-function test replacement (LFTR). The algorithm that we present, OSR, is simpler than previous algorithms because it relies on properties of SSA form. Finally, Section 10.7.3 discusses some of the issues that arise in choosing a specific application order for the optimizer's set of transformations.

10.7.1 Combining Optimizations

Sometimes, reformulating two distinct optimizations in a unified framework and solving them jointly can produce results that cannot be obtained by any combination of the optimizations run separately. As an example, consider the sparse simple constant propagation (SSCP) algorithm from Section 9.3.6. It assigns a lattice value to the result of each operation in the SSA form of the program. When it halts, it has tagged every definition with a lattice value that is either $\top$, $\bot$, or a known constant.

A final tag of $\top$ shows that the value relies on an uninitialized variable or it occurs in an unreachable block.

```
CFGWorkList ← { edges leaving $n_0$ }
SSAWorkList ← Ø

for each edge e in the CFG do
    mark e as unexecuted

for each def and each use, x, in the procedure do
    Value(x) ← ⊤

while (CFGWorkList ≠ Ø or SSAWorkList ≠ Ø) do
    if CFGWorkList ≠ Ø then
        remove an edge e = (m,n) from CFGWorkList
        if e is marked as unexecuted then
            mark e as executed
            EvaluateAllPhisInBlock((m,n))
            if no other edge entering n is marked as executed then
                if n is an assignment then
                    EvaluateAssign(n)
                    let o be n's CFG successor
                    add (n,o) to CFGWorkList
                else EvaluateConditional(n)

    if SSAWorkList ≠ Ø then
        remove an edge e = (s,d) from SSAWorkList
        c ← CFG node that uses d
        if any edge entering c is marked as executed then
            if d is a $\phi$ function argument then
                EvaluatePhi((s,d))
            else if c is an assignment then
                EvaluateAssign(c)
            else EvaluateConditional(c)
```

■ **FIGURE 10.8** Sparse Conditional Constant Propagation.

SSCP assigns a lattice value to the operand used by a conditional branch. If the value is ⊥, then either branch target is reachable. If the value is neither ⊥ nor ⊤, then the operand must have a known value and the compiler can rewrite the branch with a jump to one of its two targets, simplifying the CFG. Since this removes an edge from the CFG, it may make the block that was the branch target unreachable. Constant propagation can ignore any effects of an unreachable block. SSCP has no mechanism to take advantage of this knowledge.

We can extend the SSCP algorithm to capitalize on these observations. The resulting algorithm, called *sparse conditional constant propagation* (SCCP), appears in Figs. 10.8 to 10.10.

```
EvaluateAssign(m)            /* m is a CFG node */
    for each value y used by the expression in m do
        let (x, y) be the SSA edge that supplies y
        Value(y) $\leftarrow$ Value(x)

    let d be the name of the value produced by m
    if Value(d) $\neq \perp$ then
        v $\leftarrow$ evaluation of m over lattice values
        if v $\neq$ Value(d) then
            Value(d) $\leftarrow$ v
            for every SSA edge (d, u) do
                add (d, u) to SSAWorklist

EvaluateConditional(m)       /* m is a CFG node */
    let (s,d) be the SSA edge referenced in m
    if Value(d) $\neq \perp$ then
        if Value(d) $\neq$ Value(s) then
            Value(d) $\leftarrow$ Value(s)
            if Value(d) $= \perp$ then
                for each CFG edge (m, n) do
                    add (m, n) to CFGWorkList
            else
                let (m, n) be the CFG edge that matches Value(d)
                add (m, n) to CFGWorkList
```

■ **FIGURE 10.9** Evaluating Assignments and Conditionals.

In concept, SCCP operates in a straightforward way. It initializes the data structures. It iterates over two graphs, the CFG and the SSA graph. It propagates reachability information on the CFG and value information on the SSA graph. It halts when the value information reaches a fixed point; because the constant propagation lattice is so shallow, it halts quickly. Combining these two kinds of information, SCCP can discover both unreachable code and constant values that the compiler simply could not discover with any combination of SSCP and unreachable code elimination.

To simplify the explanation of SCCP, we assume that each CFG node, n, holds one statement plus some optional ϕ-functions. If n has one predecessor, it holds an assignment or a branch. If n has multiple predecessors, it may also have ϕ-functions before the assignment or branch.

In detail, SCCP is more complex than either SSCP or unreachable code elimination. Using two graphs introduces more bookkeeping. Making the flow of values depend on reachability adds work to the algorithm. The result is a powerful but complex algorithm.

```
EvaluatePhi((s, d))              /* (s, d) is an SSA graph edge */
    let p be the $\phi$ function that uses d
    EvaluateOperands(p)
    EvaluateResult(p)

EvaluateAllPhisInBlock((m, n))   /* (m, n) is a CFG edge */
    for each $\phi$ function p in block n do
        EvaluateOperands(p)
    for each $\phi$ function p in block n do
        EvaluateResult(p)

EvaluateOperands(phi)
    let w be the name defined by $\phi$ function phi
    if Value(w) $\neq \bot$ then
        for each parameter p of $\phi$ function phi do
            let c be the CFG edge corresponding to p
            let (x,y) be the SSA edge whose use is in p
            if c is marked as executed then
                Value(x) $\leftarrow$ Value(y)

EvaluateResult(phi)
    let x be the name defined by $\phi$ function phi
    if Value(x) $\neq \bot$ then
        v $\leftarrow$ evaluation of phi over lattice values
        if Value(x) $\neq$ v then
            Value(x) $\leftarrow$ v
            for each SSA graph edge (x, y) do
                add (x, y) to SSAWorkList
```

■ **FIGURE 10.10** Evaluating ϕ Functions.

To start, the algorithm initializes each *Value* field to $\top$ and marks each CFG edge as "unexecuted." It initializes a worklist for CFG edges to hold the set of edges that leave the procedure's entry node, n_0. It initializes a worklist for SSA edges to the empty set.

In this discussion, a block is *reachable* if and only if some CFG edge that enters it is marked as executable.

After initialization, the algorithm repeatedly picks an edge from one of the worklists and processes that edge. For a CFG edge (m, n), SCCP determines if (m, n) is marked as executed. If so, SCCP takes no further action for (m, n). If (m, n) is marked as unexecuted, then SCCP marks it as executed and evaluates all of the ϕ-functions at the start of block n. Next, SCCP checks if block n has already been entered from another edge. If not, then SCCP evaluates the assignment or conditional branch in n. This processing may add edges to either worklist.

For an SSA edge, the algorithm first checks if the destination block is reachable. If the block is reachable, SCCP calls one of *EvaluatePhi*, *EvaluateAssign*, or *EvaluateConditional*, based on the kind of operation that uses the SSA name. When SCCP must evaluate an assignment or a conditional over the lattice of values, it follows the same scheme used in SSCP, discussed in Section 9.3.6 on page 493. Each time the lattice value for a definition changes, all the uses of that name are added to the SSA worklist.

Because SCCP only propagates values into blocks that it has already proved executable, it avoids processing unreachable blocks. Because each value propagation step is guarded by a test on the executable flag for the entering edge, values from unreachable blocks do not flow out of those blocks. Thus, values from unreachable blocks have no role in setting the lattice values in other blocks.

After the propagation step, a final pass is required to replace operations that have operands with *Value* tags other than $\bot$. The final pass can specialize many of these operations. It should also rewrite branches that have known outcomes with the appropriate jump operations. A subsequent pass can remove the unreachable code (see Section 10.2.3). The algorithm cannot rewrite the code until the propagation completes.

Subtleties in Evaluating and Rewriting Operations

Some subtle issues arise in modeling individual operations. For example, if the algorithm encounters a multiply operation with operands $\top$ and $\bot$, it might conclude that the operation produces $\bot$. Doing so, however, is premature. Subsequent analysis might lower the $\top$ to the constant 0, so that the multiply produces a value of 0. If SCCP uses the rule $\top \times \bot \rightarrow \bot$, it introduces the potential for nonmonotonic behavior—the multiply's value might follow the sequence $\top, \bot, 0$, which would increase the running time of SCCP. Equally important, it might incorrectly drive other values to $\bot$ and cause SCCP to miss opportunities for improvement.

To address this, SCCP should use three rules for multiplies that involve $\bot$, as follows: $\top \times \bot \rightarrow \top$, $\alpha \times \bot \rightarrow \bot$ for $\alpha \neq \top$ and $\alpha \neq 0$, and $0 \times \bot \rightarrow 0$. This same effect occurs for any operation for which the value of one argument can completely determine the result. Other examples include a shift by more than the word length, a logical AND with zero, and a logical OR with all ones.

Some rewrites have unforeseen consequences. For example, replacing $4 \times s$, for nonnegative s, with a shift replaces a commutative operation with a noncommutative operation. If the compiler subsequently tries to rearrange expressions using commutativity, this early rewrite forecloses an opportunity.

This kind of interaction can have noticeable effects on code quality. To choose when the compiler should convert $4 \times s$ into a shift, the compiler writer must consider the order in which optimizations will be applied.

Effectiveness

SCCP can find constants that the SSCP algorithm cannot. Similarly, it can discover unreachable code that no combination of the algorithms in Section 10.2 can discover. It derives its power from combining reachability analysis with the propagation of lattice values. It can eliminate some CFG edges because the lattice values are sufficient to determine which path a branch takes. It can ignore SSA edges that arise from unreachable operations (by initializing those definitions to $\top$) because those operations will be evaluated if the block becomes marked as reachable. The power of SCCP arises from the interplay between these analyses—constant propagation and reachability.

If reachability did not affect the final lattice values, then the same effects could be achieved by performing constant propagation (and rewriting constant-valued branches as jumps) followed by unreachable-code elimination. If constant propagation played no role in reachability, then the same effects could be achieved by the other order—unreachable-code elimination followed by constant propagation. The power of SCCP to find simplifications beyond those combinations comes precisely from the fact that the two optimizations are interdependent.

10.7.2 **Strength Reduction**

Operator strength reduction is a transformation that replaces a series of expensive ("strong") operations with a series of inexpensive ("weak") operations that compute the same values. The classic example replaces integer multiplications based on a loop index with equivalent additions. This particular case arises routinely from the expansion of array and structure addresses in loops. Fig. 10.11(a) shows the ILOC that might be generated for the following loop:

```
sum ← 0
for i ← 1 to 100
    sum ← sum + a(i)
```

The code is in semipruned SSA form; the purely local values (r_1, r_2, r_3, r_4, and r_5) have neither subscripts nor ϕ-functions. Notice how the reference to a(i) expands to four operations—the subI, multI, and addI that compute (i-1) × 4 + @a and the load that defines r_4.

```
     loadI  0          ⇒ r_s0
     loadI  1          ⇒ r_i0
     loadI  100        ⇒ r_100
l_1: phi    r_i0,r_i2  ⇒ r_i1
     phi    r_s0,r_s2  ⇒ r_s1
     subI   r_i1,1     ⇒ r_1
     multI  r_1,4      ⇒ r_2
     addI   r_2,@a     ⇒ r_3
     load   r_3        ⇒ r_4
     add    r_s1,r_4   ⇒ r_s2
     addI   r_i1,1     ⇒ r_i2
     cmp_LE r_i2,r_100 ⇒ r_5
     cbr    r_5        → l_1,l_2
l_2: ...
```

(a) Original Code

```
     loadI  0          ⇒ r_s0
     loadI  @a         ⇒ r_t6
     addI   r_t6,396   ⇒ r_lim
l_1: phi    r_t6,r_t8  ⇒ r_t7
     phi    r_s0,r_s2  ⇒ r_s1
     load   r_t7       ⇒ r_4
     add    r_s1,r_4   ⇒ r_s2
     addI   r_t7,4     ⇒ r_t8
     cmp_LE r_t8,r_lim ⇒ r_5
     cbr    r_5        → l_1,l_2
l_2: ...
```

(b) Strength-Reduced Code

■ **FIGURE 10.11** Strength Reduction Example.

For each iteration, this sequence of operations computes the address of a(i) from scratch as a function of the loop index variable i. Consider the sequences of values taken on by r_{i_1}, r_1, r_2, and r_3.

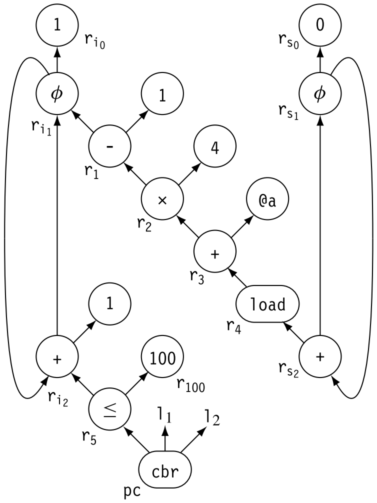
```
r_i1: { 1, 2, 3, ..., 100 }
r_1:  { 0, 1, 2, ..., 99 }
r_2:  { 0, 4, 8, ..., 396 }
r_3:  { @a, @a+4, @a+8, ..., @a+396 }
```

The only purpose for r_1, r_2, and r_3 is to compute the address for the load operation. If the code could compute each value of r_3 from the preceding one, it could eliminate the operations that define r_1 and r_2. Of course, r_3 would then need an initialization and an update; it would also need a ϕ-function at l_1 and, possibly, at l_2.

Panel (b) shows the code after strength reduction, LFTR, and dead-code elimination. It computes those values formerly in r_3 directly into r_{t_7} and uses r_{t_7} in the load operation. The end-of-loop test, which used r_{i_2} in the original code, has been modified to use r_{t_8}. This makes the computations of r_1, r_2, r_3, r_{i_0}, r_{i_1}, and r_{i_2} all dead. They have been removed to produce the final code. Now, the loop contains just five operations, ignoring ϕ-functions, while the original code contained eight. (Translation out of SSA form rewrites the ϕ-functions as copy operations; many of those copies can be coalesced.)

```
     loadI  0         ⇒ rs0
     loadI  1         ⇒ ri0
     loadI  100       ⇒ r100
l1:  phi    ri0,ri2   ⇒ ri1
     phi    rs0,rs2   ⇒ rs1
     subI   ri1,1     ⇒ r1
     multI  r1,4      ⇒ r2
     addI   r2,@a     ⇒ r3
     load   r3        ⇒ r4
     add    rs1,r4    ⇒ rs2
     addI   ri1,1     ⇒ ri2
     cmp_LE ri2,r100  ⇒ r5
     cbr    r5        → l1,l2
l2:  ...
```

(a) Example in SSA Form



(b) Corresponding SSA Graph

■ **FIGURE 10.12** Relating SSA in ILOC to the SSA Graph.

If a multI is more expensive than an addI, the savings will be larger. Historically, the high cost of a multiply justified strength reduction. However, even if multiplication and addition have equal costs, the strength-reduced form of the loop may be preferred because it creates a better code shape for later phases of the compiler. In particular, if the target machine has an autoincrement address mode, then the addI operation in the loop often can be folded into a load or store. This option does not exist with the multiply.

The rest of this section presents a simple algorithm for operator strength reduction (OSR), followed by a scheme for LFTR that shifts end-of-loop tests away from variables that would otherwise be dead. OSR operates on the SSA form of the code, considered as a graph. Fig. 10.12 shows the code for our example in SSA form alongside its SSA graph.

Background

Region constant
A value that does not vary within a given loop is a *region constant* for that loop.

Induction variable
A value that increases or decreases by a constant amount in each iteration of a loop is an *induction variable*.

Strength reduction looks for contexts in which an operation, such as a multiply, executes inside a loop and its operands are (1) a value that does not vary in that loop, called a *region constant*, and (2) a value that varies systematically from iteration to iteration, called an *induction variable*. When it finds this situation, it creates a new induction variable to compute the same sequence of values in a more efficient way. The restrictions on the form of the multiply operation's operands ensure that this new induction variable can be computed using additions, rather than multiplications.

An operation that the algorithm can reduce in this way is called a *candidate operation*. To simplify the presentation of OSR, we only consider candidate operations that have one of the formats shown in the margin, where c is a region constant and i is an induction variable. The key to finding and reducing candidate operations is efficient identification of region constants and induction variables. The restriction to these five forms is critical.

```
x ← c × i
x ← i × c
x ← c + i
x ← i + c
x ← i - c
```

Candidate Operations

A region constant can either be a literal constant, such as 10, or a loop-invariant value, that is, one not modified inside the loop. With the code in SSA form, the compiler can determine if an argument is loop invariant by checking the location of its sole definition—the block that contains the definition must dominate the entry to the loop that defines the induction variable. OSR can check both of these conditions in constant time. Performing constant propagation and lazy code motion before strength reduction may expose more region constants.

Intuitively, an induction variable is one whose values in the loop form an arithmetic progression. For the purposes of this algorithm, we can use a much more specific and restricted definition: an induction variable is a strongly connected component (SCC) of the SSA graph in which each operation that updates its value is one of (1) an induction variable plus a region constant, (2) an induction variable minus a region constant, (3) a ϕ-function, or (4) a register-to-register copy from another induction variable. While this definition is much less general than conventional definitions, it is sufficient to enable the OSR algorithm to find and reduce candidate operations. To identify induction variables, OSR finds SCCs in the SSA graph and iterates over each SCC to determine if each operation in each SCC is of one of these four types.

Because OSR defines induction variables in the SSA graph and region constants relative to a loop in the CFG, the test to determine if a value is constant relative to the loop containing a specific induction variable is complicated. Consider an operation *o* of the form x ← i × c, where i is an induction variable. For *o* to be a candidate for strength reduction, c must be a region constant with respect to the outermost loop in which i varies. To test whether c has this property, OSR must relate the SCC for i in the SSA graph back to a loop in the CFG. To simplify this test, OSR will add a field to each SSA node to hold its "header."

OSR finds the SSA graph node with the lowest reverse postorder number in the SCC that defines i. It considers this node to be the header of the SCC and records that fact in the header field of each node of the SCC. (Any node in the SSA graph that is not part of an induction variable has its header field set to *null*.) In SSA form, the induction variable's header is the ϕ-function at the start of the outermost loop in which it varies. In an operation x ← i × c, where i is an induction variable, c is a region constant if the CFG block that contains its definition dominates the CFG block that contains i's header. This condition ensures that c is invariant in the outermost loop in which i varies. To perform this test, the SSA construction must produce a map from each SSA node to the CFG block where it originated.

The header field plays a critical role in determining whether or not an operation can be reduced. When OSR encounters an operation x ← y × z, it can determine if y is an induction variable by following the SSA graph edge to y's definition and inspecting its header field. A *null* header field indicates that y is not an induction variable. If both y and z have *null* header fields, the operation cannot be strength reduced.

If one of y or z has a nonnull header field, then OSR can use that header to determine if the other operand is a region constant. Assume y's header is not null. OSR looks in the SSA-to-CFG map, indexed by y's header, to find the entry to the outermost loop where y varies. If the CFG block containing z's definition dominates the CFG block of y's header, then z is a region constant relative to the induction variable y.

The Algorithm

To perform strength reduction, OSR must examine each operation and determine if one of its operands is an induction variable and the other is a region constant. If the operation meets these criteria, OSR can reduce it by creating a new induction variable that computes the needed values and replacing the operation with a register-to-register copy from this new induction variable. (It should avoid creating duplicate induction variables.)

Based on the preceding discussion, we know that OSR can identify induction variables by finding SCCs in the SSA graph. It can discover a region constant by examining the value's definition. If the definition results from an immediate operation, or its CFG block dominates the CFG block of the induction variable's header, then the value is a region constant. The key is combining these ideas into an efficient algorithm.

OSR is built on Tarjan's strongly connected region finder. As shown in Fig. 10.13, *OSR* takes an SSA graph as its argument and repeatedly applies the strongly connected region finder, *DFS*, to it. (This process stops when *DFS* has visited every node in *G*.)

DFS performs a depth-first search of the SSA graph. It assigns each node a number that corresponds to the order in which *DFS* visits the node. It pushes each node onto a stack and labels the node with the lowest depth-first number on a node that can be reached from its children. When *DFS* returns from

```
OSR(G)
    nextNum ← 0
    while there is an unvisited n ∈ G do
        DFS(n)

DFS(n)
    n.Num ← nextNum++
    n.Visited ← true
    n.Low ← n.Num
    push(n)
    for each operand o of n do
        if o.Visited = false then
            DFS(o)
            n.Low ← min(n.Low, o.Low)
        if o.Num < n.Num and
            o is on the stack then
            n.Low ← min(n.Low, o.Num)
    if n.Low = n.Num then
        SCC ← ∅
        repeat until x = n do
            x ← pop( )
            SCC ← SCC ∪ { x }
        Process(SCC)
```

```
Process(N)
    if N has only one member n then
        if n is a "candidate" then
            Replace(n, iv, rc)
        else n.Header ← null
    else ClassifyIV(N)

ClassifyIV(N)
    IsIV ← true
    for each node n ∈ N do
        if n is not a valid update for
            an induction variable then
            IsIV ← false
    if IsIV then
        header ← n ∈ N with the
            lowest RPO number
        for each node n ∈ N do
            n.Header ← header
    else
        for each node n ∈ N do
            if n is a "candidate" then
                Replace(n, iv, rc)
            else n.Header ← null
```

■ **FIGURE 10.13** Operator Strength Reduction Algorithm.

processing the children, if the lowest node reachable from *n* has *n*'s number, then *n* is the header of an SCC. *DFS* pops nodes off the stack until it reaches *n*; all of those nodes are members of the SCC.

DFS removes SCCs from the stack in an order that simplifies the rest of *OSR*. When an SCC is popped from the stack and passed to *Process*, *DFS* has already visited all of its children in the SSA graph. If we interpret the SSA graph so that its edges run from uses to definitions, as shown in the SSA graph in Fig. 10.12, then candidate operations are encountered only after their operands have been passed to *Process*. When *Process* encounters an operation that is a candidate for strength reduction, its operands have already been classified. Thus, *Process* can examine operations, identify candidates, and invoke *Replace* to rewrite them in strength-reduced form during the depth-first search.

```
x ← c × i
x ← i × c
x ← c + i
x ← i + c
x ← i - c
```

Candidate Operations

DFS passes each SCC to *Process*. If the SCC consists of a single node *n* that has the form of a candidate operation, shown in the margin, *Process* passes *n* to *Replace*, along with its induction variable, *iv*, and its region

When *Process* identifies *n* as a candidate operation, it finds both the induction variable, *iv*, and the region constant, *rc*.

```
Replace(n, iv, rc)
    result ← Reduce(n.op, iv, rc)
    replace n with a copy from result
    n.header ← iv.header

Reduce(op, iv, rc)
    result ← Lookup(op, iv, rc)
    if result is "not found" then
        result ← NewName()
        Insert(op, iv, rc, result)

        newDef ← Clone(iv, result)
        newDef.header ← iv.header

        for each operand o of newDef do
            if o.header = iv.header then
                rewrite o with Reduce(op, o, rc)

            else if op is × or newDef.op is φ then
                replace o with Apply(op, o, rc)
    return result

Apply(op, o1, o2)
    result ← Lookup(op, o1, o2)
    if result is "not found" then
        if o1 is an induction variable and
            o2 is a region constant then
            result ← Reduce(op, o1, o2)

        else if o2 is an induction variable and
            o1 is a region constant then
            result ← Reduce(op, o2, o1)

        else
            result ← NewName()
            Insert(op, o1, o2, result)

            find block b dominated by the
                definitions of o1 and o2

            create "op o1, o2 ⇒ result"
                at the end of b and set its
                header to null

    return result
```

■ **FIGURE 10.14** Algorithm for the Rewriting Step.

constant, *rc*. *Replace* rewrites the code, as described in the next section. If the SCC contains multiple nodes, *Process* passes the SCC to *ClassifyIV* to determine whether or not it is an induction variable.

ClassifyIV examines each node in the SCC to check it against the set of valid updates for an induction variable. If all the updates are valid, the SCC is an induction variable, and *Process* sets each node's header field to contain the node in the SCC with the lowest reverse postorder number. If the SCC is not an induction variable, *ClassifyIV* revisits each node in the SCC to test it as a candidate operation, either passing it to *Replace* or setting its header to show that it is not an induction variable.

Rewriting the Code

The remaining piece of *OSR* implements the rewriting step. Both *Process* and *ClassifyIV* call *Replace* to perform the rewrite. Fig. 10.14 shows the code for *Replace* and its support functions *Reduce* and *Apply*.

Replace takes three arguments, an SSA graph node *n*, an induction variable *iv*, and a region constant *rc*. The latter two are operands of *n*. *Replace* calls *Reduce* to rewrite the operation represented by *n*. Next, it replaces *n* with a copy operation from the result produced by *Replace*. It sets *n*'s header field, and returns.

Reduce and *Apply* do most of the work. They use a hash table to avoid inserting duplicate operations. Since *OSR* works on SSA names, a single global hash table suffices. It can be initialized in *OSR* before the first call to *DFS*. *Insert* adds entries to the hash table; *Lookup* queries the table.

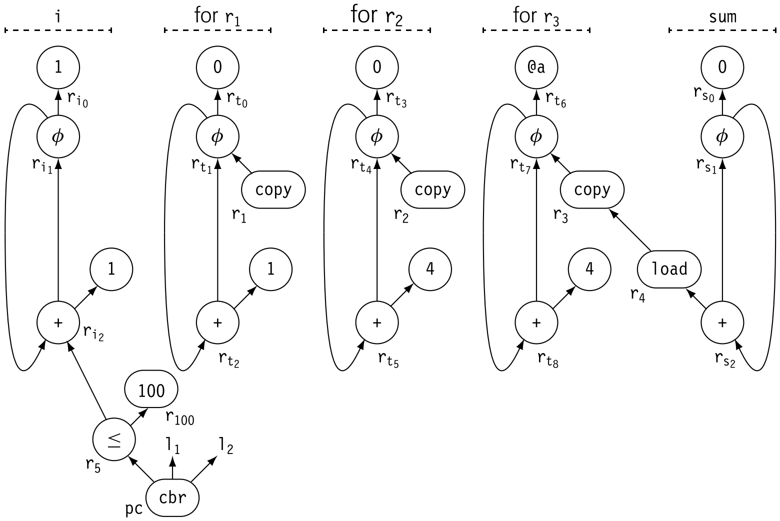
The plan for *Reduce* is simple. It takes an opcode and its two operands and either creates a new induction variable to replace the computation or returns the name of an induction variable previously created for the same combination of opcode and operands. It consults the hash table to avoid duplicate work. If the desired induction variable is not in the hash table, it creates the induction variable in a two-step process. First, it calls *Clone* to copy the definition for *iv*, the induction variable in the operation being reduced. Next, it recurs on the operands of this new definition.

These operands fall into two categories. If the operand is defined inside the SCC, it is part of *iv*, so *Reduce* recurs on that operand. This forms the new induction variable by cloning its way around the SCC of the original induction variable *iv*. An operand defined outside the SCC must be either the initial value of *iv* or a value by which *iv* is incremented. The initial value must be a ϕ-function argument from outside the SCC; *Reduce* calls *Apply* on each such argument. *Reduce* can leave an induction-variable increment alone, unless the candidate operation is a multiply. For a multiply, *Reduce* must compute a new increment as the product of the old increment and the original region constant *rc*. It invokes *Apply* to generate this computation.

Apply takes an opcode and two operands, locates an appropriate point in the code, and inserts that operation. It returns the new SSA name for the result of that operation. A few details need further explanation. If this new operation is, itself, a candidate, *Apply* invokes *Reduce* to handle it. Otherwise, *Apply* gets a new name, inserts the operation, and returns the result. (If both *o1* and *o2* are constant, *Apply* can evaluate the operation and insert an immediate load.) *Apply* locates an appropriate block for the new operation using dominance information. Intuitively, the new operation must go into a block dominated by the blocks that define its operands. If one operand is a constant, *Apply* can duplicate the constant in the block that defines the other operand. Otherwise, both operands must have definitions that dominate the header block, and one must dominate the other. *Apply* can insert the operation immediately after this later definition.

Back to the Example

Consider what happens when *OSR* encounters the example in Fig. 10.12. Assume that it begins with the node labeled $\mathtt{r}_{\mathtt{s}_2}$ and that it visits left children before right children. It recurs down the chain of operations that define $\mathtt{r}_4$,



■ **FIGURE 10.15** Transformed SSA Graph for the Example.

r_3, r_2, r_1, and r_{i_1}. At r_{i_1}, it recurs on r_{i_2} and then r_{i_0}. It finds the two single-node SCCs that contain the literal constant one. Neither is a candidate, so *Process* marks them as noninduction variables by setting their headers to *null*.

The first nontrivial SCC that *DFS* discovers contains r_{i_1} and r_{i_2}. All the operations are valid updates for an induction variable, so *ClassifyIV* marks each node as an induction variable and sets its header field to point to r_{i_1}, the node with the lowest depth-first number in the SCC.

Now, *DFS* returns to the node for r_1. Its left child is an induction variable and its right child is a region constant, so it invokes *Reduce* to create an induction variable. In this case, r_1 is r_{i_1} - 1, so the induction variable has an initial value equal to one less than the initial value of the old induction variable, or zero. The increment is the same. Fig. 10.15 shows the SCC that *Reduce* and *Apply* create, under the label "for r_1." Finally, the definition of r_1 is replaced with a copy operation, $r_1 \leftarrow r_{t_1}$. The copy operation is marked as an induction variable.

Next, *DFS* finds the SCC that consists of the node for r_2. *Process* classifies it as a candidate because its left operand (the copy that now defines r_1) is an induction variable and its right operand is a region constant. *Process* invokes *Replace* to create an induction variable with the value $r_1 \times 4$. *Reduce* and *Apply* clone the induction variable for r_1, adjust the increment to account for the multiply, and add a copy to r_2.

DFS next passes the node for r_3 to *Process*. This creates another induction variable with `@a` as its initial value and copies its value to r_3.

Process handles the `load`, followed by the SCC that computes the sum. It finds that none of these operations are candidates.

Finally, *OSR* invokes *DFS* on the unvisited node for the `cbr`. *DFS* visits the comparison, the previously marked induction variable, and the constant 100. No further reductions occur.

The SSA graph in Fig. 10.15 shows all of the induction variables created by this process. The induction variables labeled "for r_1" and "for r_2" are dead. The induction variable for `i` would be dead, except that the end-of-loop test still uses it. To eliminate this induction variable, the compiler can apply LFTR to transfer the test to the induction variable for r_3.

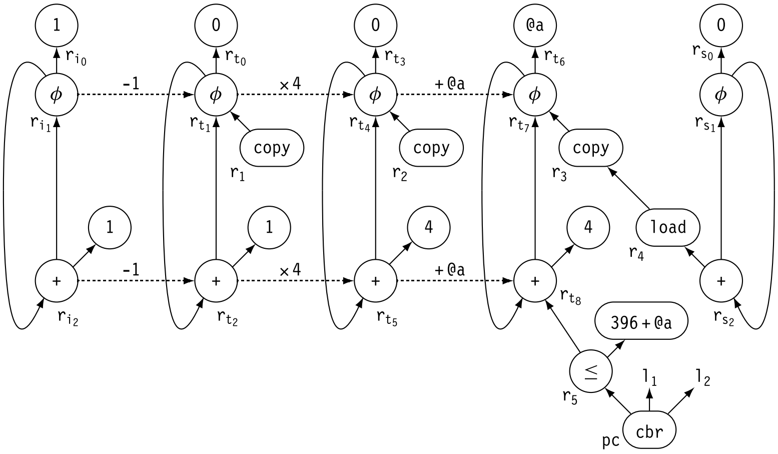
Linear-Function Test Replacement

Strength reduction often eliminates all uses of an induction variable except for an end-of-loop test. In that case, the compiler may be able to rewrite the end-of-loop test to use another induction variable found in the loop. If the compiler can remove this last use, it can eliminate the original induction variable as dead code. This transformation is called linear-function test replacement (LFTR).

To perform LFTR, the compiler must (1) locate comparisons that rely on otherwise unneeded induction variables, (2) locate an appropriate new induction variable that the comparison could use, (3) compute the correct region constant for the rewritten test, and (4) rewrite the code. Having LFTR cooperate with *OSR* can simplify all of these tasks to produce a fast, effective transformation.

The operations that LFTR targets compare the value of an induction variable against a region constant. *OSR* examines each operation in the program to determine if it is a candidate for strength reduction. It can easily and inexpensively build a list of all the comparison operations that involve induction variables. After *OSR* finishes its work, LFTR should revisit each of these comparisons. If the induction-variable argument of a comparison was strength reduced by *OSR*, LFTR should retarget the comparison to use the new induction variable.

To facilitate this process, *Reduce* can record the arithmetic relationship it uses to derive each new induction variable. It can insert a special LFTR edge from each node in the original induction variable to the corresponding node in its reduced counterpart and label it with the operation and region constant of the candidate operation responsible for creating that induction variable.



■ **FIGURE 10.16** Example After Linear Function Test Replacement.

In Fig. 10.16 these additional edges appear as dashed lines. The sequence of reductions in the example create a chain of labeled edges. Starting from the original induction variable, we find the labels -1, x4, and +@a.

When LFTR finds a comparison that should be replaced, it can follow the edges from its induction-variable argument to the final induction variable that resulted from a chain of one or more reductions. The comparison should use this induction variable with an appropriate new region constant.

The labels on the LFTR edges describe the transformation that must be applied to the original region constant to derive the new region constant. In the example, the trail of edges leads from r_{i_2} to r_{t_8} and produces the value (100 - 1) × 4 + @a for the transformed test. Fig. 10.16 shows the edges and the rewritten test.

This version of LFTR is simple, efficient, and effective. It relies on close collaboration with *OSR* to identify comparisons that might be retargeted and to record the reductions as it applies them. Using these two data structures, LFTR can find comparisons to retarget, find the appropriate place to retarget them, and find the necessary transformation for the comparison's constant argument.

10.7.3 **Choosing an Optimization Sequence**

Optimization sequence
a set of optimizations and an order for their application

The effectiveness of an optimizer on any given code depends on the sequence of optimizations that it applies to the code—both the specific transformations that it uses and the order in which it applies them. Traditional optimizing compilers have offered the user the choice of several sequences (e.g., `-O`, `-O1`, `-O2`, ...) that provide a tradeoff between compile time and optimization. Increased optimization effort, however, does not guarantee improvement.

The optimization sequence problem arises because the effectiveness of any given transformation depends on several factors.

1. Does the opportunity that the transformation targets appear in the code? If not, the transformation will have no effect.
2. Has a prior transformation hidden or obscured that opportunity? For example, LVN can convert `2 × a` into a shift operation. However, multiply is commutative and shift is not. Any transformation that needs commutativity to effect its improvement might see opportunities vanish from prior application of LVN.
3. Has any other transformation already eliminated the inefficiency? Transformations have overlapping and idiosyncratic effects; for example, LVN achieves some of the effects of global constant propagation and loop unrolling achieves some of the same effects as superblock cloning. The compiler writer might still include both transformations for their nonoverlapping effects.

The interactions between transformations makes it difficult to predict the improvement from the application of any single transformation or any sequence of transformations.

Researchers have looked at automated techniques to find good optimization sequences. The approaches vary in granularity and in technique. The various systems look for sequences at the block level, at the source-file level, and at the program level. Most of these systems use some kind of search over the space of optimization sequences.

In general, these techniques are too expensive to be practical for individual programs because the space of potential sequences is too large. For example, if the compiler chooses a sequence of 10 transformations from a set of 15, it can generate 10^{15} possible sequences—an impractically large number to explore. Thus, compilers that search for good sequences use heuristic techniques to sample subsets of the search space. In general, these techniques fall into three categories: (1) genetic algorithms adapted to act as intelligent searches, (2) randomized search algorithms, and (3) statistical machine learning techniques. All three approaches have shown promise.

Despite the huge search spaces, well-tuned search algorithms can find good optimization sequences with 100 to 200 probes of the search space. While that number is not yet practical, further refinement may reduce the number of probes to a practical level.

In this context, a *good* sequence is one that produces results within 5 percent of the best results.

The primary interesting application of these techniques is to derive the sequences used by the compiler's command line flags, such as `-O2`. The compiler writer can use an ensemble of representative applications to discover good general sequences and then apply those sequences as the compiler's default sequences. A more aggressive approach, used in several systems, is to derive a handful of good sequences for different application ensembles and have the compiler try each of those sequences and retain the best result.

10.8 SUMMARY AND PERSPECTIVE

The design and implementation of an optimizing compiler is a complex undertaking. This chapter has introduced a conceptual framework for thinking about transformations—the taxonomy of effects. Each category in the taxonomy is represented by several examples, either in this chapter or elsewhere in the book.

The challenge for the compiler writer is to select a set of transformations that work well together to produce good code—code that meets the user's needs. The taxonomy presented in this chapter is a framework for thinking about that decision. The specific transformations implemented in a compiler determine, to a large extent, the kinds of programs for which it will produce good code.

CHAPTER NOTES

While the algorithms presented in this chapter are modern, many of the basic ideas were well known in the 1960s and 1970s. Dead-code elimination, code motion, strength reduction, and redundancy elimination are all described by Allen [12] and by Cocke and Schwartz [98]. A number of survey papers provide overviews of the state of the field at different points in time [17,29,31,328]. Books by Morgan [277] and Muchnick [279] both discuss the design, structure, and implementation of optimizing compilers. Wolfe [364] and Allen and Kennedy [21] focus on dependence-based analysis and transformations for vector and parallel computation.

Dead implements a mark-sweep style of dead-code elimination that was introduced by Kennedy [226,228]. It is reminiscent of the Schorr-Waite marking algorithm [319]. *Dead* is specifically adapted from the work of Cytron et al. [120, Section 7.1]. *Clean* was developed and implemented in 1992 by Rob Shillingsburg [262].

LCM improves on Morel and Renvoise's classic algorithm for partial redundancy elimination [276]. That paper inspired many improvements [88,141, 144,333]. Knoop, Rüthing, and Steffen's LCM [236] improved code placement; the formulation in Section 10.3 uses equations from Drechsler and Stadel [145]. Bodik, Gupta, and Soffa combined this approach with replication to find and remove all redundant code [47]. The DVNT algorithm is from Simpson's thesis [59,326]. It has been implemented in a number of compilers.

Hoisting appears in the Allen-Cocke catalogue as a technique for reducing code space [17]. The formulation using anticipability appears in several places, including Fischer and LeBlanc [157]. Sinking or cross-jumping is described by Wulf et al. [368].

Both peephole optimization and tail-recursion elimination date to the early 1960s. Peephole optimization was first described by McKeeman [268]. Tail-recursion elimination is older; folklore tells us that McCarthy described it at the chalkboard during a talk in 1963. Steele's thesis [335] is a classic reference for tail-recursion elimination.

Superblock cloning was introduced by Hwu et al. [211]. Loop optimizations such as unswitching and unrolling have been studied extensively [21,29]; Kennedy used unrolling to avoid copy operations at the end of a loop [225]. Cytron, Lowrey, and Zadeck present an interesting alternative to unswitching [121]. McKinley et al. give practical insight into the impact of memory optimizations on performance [101,269].

Combining optimizations, as in SCCP [358,359], often leads to improvements that cannot be obtained by independent application of the original optimizations [89,91]. Value numbering combines redundancy elimination, constant propagation, and simplification of algebraic identities [59]. LCM combines elimination of redundancies and partial redundancies with code motion [236]. Click and Cooper [91] combine Alpern's partitioning algorithm [22] with SCCP [359]. Many authors have combined register allocation and instruction scheduling [44,54,173,278,285,286,294,318]. Not all pairs of optimizations benefit from being combined [89,91].

Operator strength reduction has a rich history. One family of strength-reduction algorithms developed out of work by Allen, Cocke, and Kennedy [20,95,97,227,264]. The OSR algorithm is in this family [117]. Another family of algorithms grew out of the data-flow approach to optimization exemplified by the LCM algorithm; a number of sources give techniques in this family [138,140,142,187,220,231,237]. The version of OSR in Section 10.7.2 only reduces multiplications. Allen et al. show the reduction sequences for many other operators [20]; extending OSR to handle these cases is straightforward. A weaker form of strength reduction rewrites integer multiplies with faster operations [251].

EXERCISES

Section 10.1

1. One of the primary functions of an optimizer is to remove overhead that the compiler introduced during the translation from source language into IR.

   a. Give four examples of inefficiencies that you would expect an optimizer to improve, along with the source-language constructs that give rise to them.

   b. Give four examples of inefficiencies that you would expect an optimizer to miss. Explain why an optimizer would have difficulty improving them.

Section 10.2

2. Fig. 10.1 shows the algorithm for *Dead*. The marking pass is a classic fixed-point computation.

   a. Explain why this computation terminates.

   b. Is the fixed-point that it finds unique? Prove your answer.

   c. Derive a tight time bound for the algorithm.

3. Consider the algorithm *Clean* from Section 10.2. It removes useless control flow and simplifies the CFG.

   a. Why does the algorithm terminate?

   b. Give an overall time bound for the algorithm.

Section 10.3

4. Redundancy elimination has a variety of effects on the code that the compiler generates.

   a. How does LCM affect the demand for registers in the code being transformed? Justify your answer.

   b. How does LCM affect the size of the code generated for a procedure? (Assume that demand for registers is unchanged.)

   c. How does hoisting affect the demand for registers in the code being transformed? Justify your answer.

   d. How does hoisting affect the size of the code generated for a procedure? (Assume that demand for registers is unchanged.)

Section 10.4

5. A simple form of operator strength reduction replaces a single instance of an expensive operation with a sequence of operations that are less expensive to execute. For example, some integer multiply operations can be replaced with a sequence of shifts and adds.

   a. What conditions must hold to let the compiler safely replace an integer operation $x \leftarrow y \times z$ with a single shift operation?

b. Sketch an algorithm that replaces a multiplication of a known constant and an unsigned integer with a sequence of shifts and adds.

c. How might you fit this transformation into the LVN algorithm?

6. Both tail-call optimization and inline substitution attempt to reduce the overhead caused by the procedure linkage.

   a. Can the compiler inline a tail call? What obstacles arise? How might it work around them?

   b. Contrast the code produced from your modified inlining scheme with that produced by tail-call optimization.

Section 10.5

7. A compiler can find and eliminate redundant computations in many different ways. Among these are DVNT and LCM.

   a. Show two examples of redundancies eliminated by DVNT that cannot be found by LCM.

   b. Show an example of a redundancy that LCM finds and improves but that DVNT cannot find.

Section 10.6

8. Sketch an algorithm that renames the values and variables in a procedure so that name identity encodes value identity.

9. Superblock cloning can cause significant code growth.

   a. How might the compiler mitigate code growth in superblock cloning while retaining as much of the benefit as possible?

   Hint: Think back to the block-placement algorithm in Chapter 8.

   b. What problems might arise if the optimizer allowed superblock cloning to continue across a loop-closing branch?

This page intentionally left blank

Chapter 11

Instruction Selection

ABSTRACT

The compiler's front end and optimizer both operate on the code in its IR form. To create executable code, the compiler must rewrite the IR into the processor's instruction set. This process is called instruction selection. Because a typical processor provides multiple ways to express most computations, the selector must choose the best sequence from among multiple implementations.

This chapter introduces two distinct approaches to instruction selection. The first builds on peephole optimization, a classic late-stage code-improvement technique. The second uses tree-pattern matching algorithms to find a set of operations that implement the IR program. Both approaches are supported by reliable tools. Both have found widespread use in real compilers.

KEYWORDS

Instruction Selection, Tree-Pattern Matching, Peephole Optimization

11.1 INTRODUCTION

To translate a program from an intermediate representation, such as an abstract syntax tree or a low-level linear code, into executable form, the compiler must map each IR construct into an equivalent construct in the target processor's instruction set. Depending on the relative levels of abstraction in the IR and the target machine's instruction set architecture (ISA), this translation can involve elaborating details that are hidden in the IR program or it can involve combining multiple IR operations into a single machine instruction. The specific choices that the compiler makes have a direct impact on the overall efficiency of the compiled code.

The complexity of instruction selection derives from the large number of ways that a typical ISA can implement even simple operations. In the 1970s, the DEC PDP/11 had a compact instruction set; thus, a good compiler such as the BLISS-11 compiler could perform instruction selection with a simple hand-coded pass. As processor ISAs expanded, the number of possible ways to implement a given sequence of IR operations grew unmanageable. This explosion led to systematic approaches for instruction selection, such as those presented in this chapter, and tools that implement them.

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00017-6

Conceptual Roadmap

Instruction selection, which maps code in the compiler's IR into code in the target ISA, is a pattern-matching problem. At its simplest, the compiler could provide a single assembly code sequence for each IR operation. The resulting compiler would produce correct, albeit template-like, code that might make poor use of the target machine's resources. Better approaches consider multiple code sequences for each IR operation, along with the operation's context, to choose the sequence that has the lowest expected cost.

This chapter presents two approaches to instruction selection: the first based on peephole optimization and the second based on tree-pattern matching. The former approach translates the compiler's IR into a low-level linear IR, systematically improves that IR, and then maps the improved IR into the target machine's ISA. The latter approach represents both the compiler's IR and the target machine's ISA with tree patterns and uses pattern matching to find an implementataion of the IR program in the target machine's ISA. Each of these techniques can produce high-quality code that is tailored to the local context. Each has been incorporated into tools that take a target machine description and produce a working instruction selector.

A Few Words About Time

The instruction selector, itself, runs at compile time to translate the IR program created by the front end and optimizer into code expressed in the target machine's ISA. The code produced by selection may not be a valid assembly program; in many compilers, the postselection code still assumes an unlimited supply of virtual registers. The register allocator, which runs after the instruction selector at compile time, completes the task of mapping the code into the target ISA's register set.

Many compilers use description-based instruction selectors, based on either peephole optimization (see Section 11.3) or tree-pattern matching (see Section 11.4). With these tools the compiler writer creates, at design time, descriptions of the compiler's IR and the target ISA. At build time, a back-end generator analyzes these descriptions and produces code for the instruction selector, which is then compiled and included in the compiler. This chapter focuses on the compile-time algorithms used to perform selection.

Overview

Systematic approaches to code generation make it easier to retarget a compiler. The goal is to minimize the effort needed to port the compiler to a new processor. Ideally, the front end and the optimizer need minimal changes, and much of the back end can be reused as well. This strategy makes good

SELECTION, SCHEDULING, AND ALLOCATION

The three major processes in the back end are instruction selection, scheduling, and register allocation. All three processes have a direct impact on the quality of the generated code, and they all interact with each other.

Selection directly changes the scheduling problem. Selection dictates both the time required for an operation and the functional units on which it can execute. Scheduling might affect instruction selection. If the code generator can implement an IR operation with either of two assembly operations, and those operations use different resources, the code generator might need to understand the final schedule to ensure the best choice.

Selection interacts with register allocation in several ways. If the target processor has a uniform register set, then the instruction selector can assume an unlimited supply of registers and rely on the allocator to insert the loads and stores needed to fit the values into the register set. If, on the other hand, the target machine has complex register use rules, then the selector may need to pay close attention to specific physical registers. This can complicate selection and predetermine some or all of the allocation decisions. In this situation, the code generator might use a coroutine to perform local register allocation during instruction selection.

Keeping selection, scheduling, and allocation separate—to the extent possible—can simplify the implementation, testing, and debugging of each process. However, since each of these processes can constrain the others, the compiler writer must be aware of the interactions to ensure that separation of these processes does not harm code quality.

use of the investment in building, debugging, and maintaining the reusable parts of the compiler.

Much of the responsibility for handling diverse targets rests on the instruction selector. A typical compiler uses a common IR for all targets and, to the extent possible, for all the source languages that it supports. It optimizes the IR based on a set of assumptions that hold true on most, if not all, target machines. Finally, it uses a back end in which the compiler writer has tried to isolate and extract the target-dependent details.

In practice, a new language may need some new operations in the IR. The goal, however, is to extend the IR, rather than to reinvent it.

While the scheduler and register allocator need target-dependent information, good design can isolate that knowledge into a concrete description of the target machine and its ISA. Such a description might include register-set sizes; a description of each operation; the number, capabilities, and operation latencies of the functional units; memory alignment restrictions; and the procedure-call convention. The algorithms for scheduling and allocation are then parameterized by those system characteristics and reused across different ISAs and systems.

Thus, the key to compiler retargetability lies in the implementation of the instruction selector. The selector consumes the compiler's IR and produces assembly code for the target machine. A retargetable instruction selector consists of a pattern-matching engine coupled to information about the IR, the ISA, and the mapping between them.

A back-end generator is sometimes called a *code-generator generator*.

If we can automate the construction of the matching engine, then specification-driven construction of instruction selectors can follow the same basic model as scanners and parsers. In those systems, the compiler writer creates a description of the syntax and the actions to be taken; she then invokes a scanner generator or parser generator to build the actual compiler component. In a similar way, the compiler writer can create a description of the target machine and invoke a *back-end generator* to construct an instruction selector. The generator runs offline at build time.

This approach moves the cost and complexity of instruction selection into the back-end generator. Just as in LR parsing, we can afford to use algorithms in the generator that require more time than the algorithms that run at compile time. In fact, tool-based instruction selectors can be extremely efficient at compile time.

A second key to compiler retargetability is to isolate machine-dependent code to the greatest extent possible. Ideally, all machine-dependent code should appear in the instruction selector, scheduler, and register allocator; unfortunately, the reality almost always falls short of this ideal. Some machine-dependent details creep, unavoidably, into earlier parts of the compiler. For example, the alignment restrictions on activation records (ARs) may differ among target machines, changing offsets for values stored in ARs. The compiler may need to represent features such as predicated execution, branch delay slots, and multiword memory operations explicitly if it is to make good use of them. Still, pushing target-dependent details into instruction selection can reduce the number of changes to other parts of the compiler that are needed to port it to a new target processor.

As a design question, the compiler writer must decide how much customization of the code should occur in the compiler's back end. Our belief is that the instruction selector should find a good local mapping from the code that the optimizer produced into the target machine's instruction set. Primary responsibility for efficiency rests with the compiler's optimizer, not its back end. Logic and experience suggest that we separate the goal of optimizing computation from the goal of mapping that computation efficiently onto the target machine.

This chapter examines two approaches used to build instruction selectors. The next section explores background issues in instruction selection and introduces the example used throughout the chapter. The two subsequent sections present different ways to apply pattern-matching techniques to transform IR sequences to assembly sequences. Section 11.3 presents an approach based on ideas from peephole optimization. Section 11.4 explores an approach based on matching tree patterns against trees. Both of these methods are description based. The compiler writer creates a description of the target ISA; a tool then constructs a selector for use at compile time. Both methods have been used in successful portable compilers.

11.2 BACKGROUND

Instruction selection rewrites the code from IR form into target machine code. If the IR and the ISA have similar levels of abstraction, then this translation may be straightforward—as with rewriting ILOC code to run on a simple RISC machine. If the IR is more abstract than the ISA, then the selector must supply additional detail—as with mapping a near-source AST onto a commodity microprocessor. Finally, if the IR is less abstract than the ISA, as with GCC's register-transfer language, then the selector may need to combine multiple IR operations into a single target machine operation.

The compiler writer's goals come into play, as well. The compiler writer must confront the tradeoff between complexity in the compiler's back end and the performance of the compiled code. It takes a more complex compiler to produce customized and optimized code than to produce slower template-like code. In an appropriate context, either approach might make sense.

Chapter 5 discussed the issues that arise when the compiler translates source code into an initial IR. Instruction selection differs from IR generation in two major ways. First, IR generation should produce a version of the program that *compiles* and *optimizes* well, while instruction selection should produce a version that *runs* well. Second, the compiler writer has control over the design of the IR, which can simplify the task of mapping source code into IR; the target ISA is a fixed language, designed by other people and bound by limited resources. Because of these differences, code generation in a compiler's back end uses more complex approaches and algorithms than does IR generation in its front end.

The complexity of instruction selection arises not from a particular methodology or algorithm, but rather, from the nature of the underlying problem—a processor typically provides multiple ways to implement an IR construct, each with its own costs and its own restrictions. The code generator must choose among these alternatives based on knowledge of operation costs and

CODE THAT RUNS WELL VERSUS OPTIMIZES WELL

The difference between generating IR early in the compiler and generating assembly code late in the compiler lies in the intended use of the code. The IR form of the program targets optimization and code generation. The assembly code version targets fast execution. To see how these different use cases drive the compiler to different code sequences, consider the following abstracted loop.

```
for i = 1 to 1000
    for j = 1 to 1000
        ... A[i,j] ...
```

Section 7.3.2 showed two ways to compute the address of `A[i,j]`, assuming that `A` is dimensioned as `A[`low_1`:`$high_1$`,`low_2`:`$high_2$`]` and stored in row-major order starting at `@A`:

(1) $\texttt{@A} + (\texttt{i} - low_1) \times len_2 \times w + (\texttt{j} - low_2) \times w$

(2) $\texttt{@A} + (\texttt{i} \times len_2 \times w) + (\texttt{j} \times w)$
$\quad - (low_1 \times len_2 \times w + low_2 \times w).$

Polynomial (1) uses fewer operations; it will execute more quickly than the latter one. Polynomial (2) separates out subexpressions that depend on `i` and `j`, plus a term that contains only constants. A good optimizer might move the `i` term out of the inner loop and evaluate the constant term at compile time. Thus, formula (1) runs well while formula (2) optimizes well.

surrounding context. Done well, this process should produce efficient code that is customized to fit well with the surrounding operations.

Specification-based tools can move most of the complexity of generating customized instruction sequences from compile time back into design time and build time. Such tools automatically construct efficient and effective instruction selectors that manage the complexity of context in both the IR and the ISA. They can simplify construction of an effective compiler back end, in much the same way that scanner generators and parser generators simplify front-end construction.

11.2.1 The Impact of ISA Design on Selection

Remember that ILOC, used for the examples, is an extremely simple case.

Much of the complexity of instruction selection arises directly from properties of a typical target machine ISA. Two factors that cause an explosion in the number of cases that the selector must consider are: duplicate mechanisms to accomplish a single task and the proliferation of address modes in memory and arithmetic operations.

Features such as these complicate instruction selection, because they add context, they add complexity, and they add choices. Automated techniques for building instruction selection arose, in large part, as a response to increasing processor complexity. By specifying the IR to ISA mapping in a more concise way and algorithmically expanding that specification into code, the tools simplify the task of building efficient and powerful selectors.

Duplicate Implementations

If the target ISA provided just one way to perform each IR operation, the compiler could simply rewrite each IR operation with the equivalent sequence of machine operations. In most contexts, however, a target machine provides multiple ways to implement each IR construct.

Consider a simple copy operation $r_i \rightarrow r_j$. Assume that the target processor uses ILOC as its native instruction set. ILOC is simple, but even it exposes the complexity of code generation. The obvious implementation of $r_i \rightarrow r_j$ uses `i2i` $r_i \Rightarrow r_j$; such a register-to-register copy is typically one of the least-expensive operations that a processor provides. However, other implementations abound. These include,

```
addI r_i,0   ⇒ r_j    subI    r_i,0 ⇒ r_j    multI   r_i,1 ⇒ r_j
divI r_i,1   ⇒ r_j    lshiftI r_i,0 ⇒ r_j    rshiftI r_i,0 ⇒ r_j
and  r_i,r_i ⇒ r_j    orI     r_i,0 ⇒ r_j    xorI    r_i,0 ⇒ r_j
```

Still more possibilities exist. If the processor has a register whose value is always 0, another set of operations works, using `add`, `sub`, `lshift`, `rshift`, `or`, and `xor`. If we consider two-operation sequences, the set is even larger.

For example, the ARM V8 has a dedicated zero register, `XZR`.

A programmer would discount most, if not all, of these alternatives. A register-to-register copy operation, such as `i2i`, is simple, fast, and obvious. An automated process, however, may need to consider all the possibilities and their costs. The fact that an ISA can accomplish an effect in multiple ways makes instruction selection harder. Even ILOC, a particularly simple ISA, has many ways to perform a copy operation.

Address Modes

The arithmetic operations listed as variants for a copy operation all assumed a register-to-register addressing model. Some ISAs provide variant operators that get one or more of their operands from memory. Other operators, such as load and store, always address memory. To improve efficiency, ISAs introduce *address modes* that encode common address computations.

RISC, CISC, AND INSTRUCTION SELECTION

Early proponents of RISC architectures suggested that RISC ISAs would lead to simpler compilers. Early RISC machines, such as the IBM 801, had many fewer address modes than contemporary CISC machines such as DEC's VAX-11 series. They featured register-to-register operations, with separate load and store operations to move data between registers and memory. By contrast, the VAX-11 offered both register and memory operands; many operations were supported in both two-address and three-address forms.

The RISC machines did simplify instruction selection. They offered fewer ways to implement a given operation. They had fewer restrictions on register use. However, their load-store architectures increased the importance of register allocation.

By contrast, CISC machines have operations that encapsulate more complex functionality into a single operation. To make effective use of these operations, the selector must recognize larger patterns over larger code fragments. This effect increases the importance of systematic instruction selection; the automated techniques described in this chapter are more important for CISC machines, but equally applicable to RISC machines.

ILOC provides multiple address modes for `load` and `store`.

Operation	Meaning
`load` $r_1 \Rightarrow r_2$	MEMORY$(r_1) \Rightarrow r_2$
`loadAI` $r_1, c_2 \Rightarrow r_3$	MEMORY$(r_1 + c_2) \Rightarrow r_3$
`loadAO` $r_1, r_2 \Rightarrow r_3$	MEMORY$(r_1 + r_2) \Rightarrow r_3$
`loadI` $c_1 \Rightarrow r_2$	$c_1 \Rightarrow r_2$

Store operations support the address-immediate (`AI`) and address-offset (`AO`) modes. Arithmetic and logical operations also support an immediate (`I`) mode that reads one operand as a literal constant embedded in the instruction stream.

Real processors often support a larger and more complex set of address modes. Examples include memory-to-memory arithmetic operators, one or two address operations that overwrite some of their input arguments, and operations with implicit arguments, such as the top of a hardware-supported runtime stack. The selector should make effective use of all the address modes that the ISA supports.

Similar issues arise in control-flow operations. Branches and jumps may support absolute addresses, program-counter relative addresses, and addresses of different bit-lengths. These operations have different lengths; they may take different numbers of cycles. Because the selection of the best form of branch or jump depends on multiple factors, including the distance and direction from source to destination, selecting branch address modes requires painstaking care.

Level of Abstraction

The individual operations in ILOC have a relatively low level of abstraction. Modern processors often provide a mix of low-level operations and more complex operations. For example:

- A string-move operation allows the code to easily specify a complex sequence that includes an implicit iterative loop.
- A procedure-call operation might automate large parts of the call sequence, including management of caller-saves registers.
- A floating-point multiply-add operation might use fewer cycles and fewer registers to compute $(r_i \times r_j) + r_k$ than the individual multiply and add operations.
- A load-multiple or store-multiple operation might move values into or out of several contiguous registers.

Operations such as these can require the instruction selector to synthesize several low-level operations into one higher-level operation.

Register Use

Processors constrain register use in idiosyncratic ways (see Section 13.2.4). Operations may expect one or more operands in designated locations. On some processors, double-precision values must be stored in a pair of single-precision floating-point registers that begin with an even-numbered register. The use of a destructive one-address or two-address operation can overwrite the value of one of its arguments, which complicates their allocation and use (see Section 4.4).

For example, on the IA-32, a 16-bit signed integer multiply reads one argument from AX and writes its 32-bit result into AX and DX.

Costs

Each operation has its own cost. Most modern machines implement simple operations, such as an addition or a shift, as single-cycle operations. Other operations, such as a load or a divide, may take longer. The latency of multiplication and division may depend on the bit patterns in the operands. The latency of a memory operation depends on many factors, including the detailed current state of the processor's memory system. The latency of a branch may depend on how well the processor can predict its outcome.

In most cases, the compiler writer wants the back end to produce code that runs quickly. However, other metrics are possible. For example, if the final code will run on a battery-powered device, the compiler might consider the typical energy consumption of each operation; individual operations consume different amounts of energy. A compiler that tries to optimize for energy may use radically different costs than would one optimizing for speed. Similarly, if code space is critical, the compiler writer might assign costs based solely on sequence length. Alternatively, the compiler writer might simply exclude from consideration all multioperation sequences that achieve the same effect as a single-operation sequence.

Since a shorter code sequence fetches fewer bytes from RAM, reducing code space may also reduce energy consumption.

While instruction selection can play an important role in determining code quality, the compiler writer must keep in mind the enormous size of the search space that the instruction selector might explore. Even moderately sized instruction sets can produce search spaces that contain millions of states. Clearly, the compiler cannot afford to explore such spaces exhaustively. The techniques that we describe explore the space of alternative code sequences in a disciplined fashion and either limit their searches or precompute enough information to make a deep search efficient.

11.2.2 Motivating Example

The discussions of instruction selection via peephole optimization and tree-pattern matching use the same example to motivate, explain, and explore the issues. In both cases, we examine how to generate code for the simple assignment statement: `a` ← `b - 2` × `c`. Fig. 11.1 shows the example in two different IRs. Panel (a) shows the statement in quadruple form; the discussion of peephole-based instruction selection starts from this IR. Panel (b) shows a low-level AST for the statement; the discussion of tree-pattern matching uses this IR.

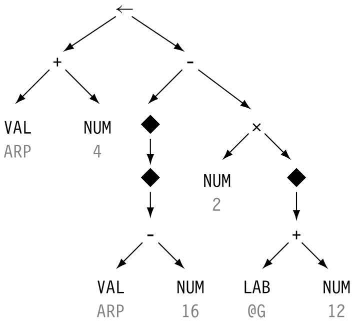
This example first appeared in Fig. 4.1 on page 163.

The example uses three variables, each with a different kind of address computation: `a` is a local variable that resides at offset 4 from the ARP; `b` is a call-by-reference parameter that resides at offset –16 from the ARP; and `c` is a global variable.

The address computation for `c` is complex. The value resides at offset 12 from a global label. Because the label resolves to a full-length address, the resolved address is stored in some global constant pool. The constant pool starts at `@CP` and the offset in the pool is given by `@G`. We assume that `@CP` is too large for an immediate field in an `add` or `sub`, but small enough for the immediate field in a `loadI`.

Op	Arg_1	Arg_2	Result
×	2	c	t_1
-	b	t_1	a

(a) Quadruples



(b) Low-Level AST

■ **FIGURE 11.1** Low-Level IRs for a ← b - 2 × c.

Because the example consists of a single source-level statement, it contains no control flow. Thus, we will assume that any condition-code values defined by the operations are unused and therefore dead.

Quadruples

Fig. 11.1(a) shows the example code expressed in classic quadruple form. Variable names appear as references, with additional detail available in the symbol table. Operations are simple three-address code. A compiler-generated temporary name is used to carry the result of the multiply into the subtraction. The order of the quadruples encodes the execution order.

This representation has little explicit detail. It assumes a symbol table that contains type and location data for the named values (see Sections 4.5 and 5.4). As we will see in Section 11.3, the peephole instruction selector will immediately expand this code to make all of the necessary details explicit and exposed in the IR code itself.

Low-Level AST

To expose enough detail for instruction selection, the AST shown in Fig. 11.1(b) has a low-level of abstraction. Several of the nodes in the tree need further explanation.

Constant values are represented by three distinct kinds of nodes:

- A NUM node represents a constant that fits into the immediate field of a three-operand immediate instruction (e.g., multI or loadAI).
- A CON node, not shown in this example, represents a constant that is small enough to fit in a loadI, but too large for a NUM.

- A LAB node represents a relocatable symbol, typically an assembly-level label used for either code or data.

The distinction between these kinds of constants is critical to instruction selection. A value in a CON or LAB node cannot appear as an immediate operand in a multI operation. A value represented by a LAB node is assumed to be too large to fit in a loadI operation; the compiler stores such values into a constant pool in memory, located at @CP.

Two other nodes in this tree have nonobvious meanings. A VAL node represents a value known to reside in a register, such as the ARP in r_{arp}, or the result of evaluating a common subexpression identified by the optimizer. A ◆ node signifies a level of indirection; its child is an address and it produces the value stored at that address.

The low-level detail in the AST allows the instruction selector to tailor its decisions to specific context. The subtrees that describe the address expressions for a, b, and c all look similar; as we shall see, they generate distinctly different code due to the specific base addresses and offsets. By tailoring the final assembly code to context, the compiler can produce efficient code for each subtree.

11.2.3 **Ad-Hoc Matching**

The compiler writer can construct an instruction selector around a handwritten, ad-hoc matcher. Such a selector would recognize individual IR constructs or small sets of related constructs and map them directly into the assembly language (ASM) of the target machine. For a tree-based IR, this approach might result in a simple postorder walk, similar to the syntax-driven translation schemes of Chapter 5. For a linear IR, the compiler writer might produce a selector that makes a linear scan over the IR and emits code for each operation.

Instruction selectors built around ad-hoc matching tend to produce uniform, template-like code. To take better advantage of the target machine's ISA requires the aggregation of context from the IR program. The selector can only make effective use of complex address modes if it builds up some representation, implicit or explicit, of the address computation; it can use that model to choose the best load or store operation. The selector can only make use of immediate-mode operations if it understands, even in limited ways, the flow of constant values through the code and the magnitudes of those values.

Finally, ad-hoc matching provides little, if any, support for retargeting the compiler to another processor. Portability is a key rationale for isolating the instruction selector both from optimization and from register allocation and instruction scheduling. A tool-based approach to specification-driven instruction selectors should simplify the task of retargeting the compiler to a new processor or system.

SECTION REVIEW

An instruction selector rewrites the IR code into the target machine's assembly code. It must, for each IR construct, choose an efficient assembly-language implementation, tailored to the surrounding context. The selector itself must run quickly.

Instruction selection is hard because of the sheer number of choices that the selector confronts. It must choose from multiple operations. It must make effective use of the target's address modes. It must model any restrictions on processor resource use and ensure that the generated code respects them. Because each choice affects the runtime efficiency of the compiled code, the selector plays a critical role in application performance.

REVIEW QUESTIONS

1. Enumerate as many ways as you can in ILOC to set register r_i to contain the integer value one.
2. Consider an operation `move` $r_i \Rightarrow r_k$ that copies four bytes from memory address r_i to memory address r_k at a cost of one more cycle than the equivalent `load` operation. What advantages might such an operation present?

11.3 SELECTION VIA PEEPHOLE OPTIMIZATION

Peephole optimization is a late-stage optimization technique developed in the 1960s and 1970s. Some compilers use peephole optimization as a paradigm for instruction selection. They use a selector that translates the IR program into a low-level IR (LLIR), applies systematic local optimization to the LLIR, and then uses matching to map the LLIR into target machine instructions. The local optimization builds up the knowledge needed to target effectively processor features such as address modes. It also systematically eliminates minor inefficiencies that optimization at a higher level of abstraction might miss.

11.3.1 Peephole Optimization

The basic premise of peephole optimization is simple: the compiler can efficiently find local improvements by examining short sequences of adjacent operations. This premise seems to contradict the earlier assertion that the optimizer has done its job well. The distinction is subtle. When peephole optimization finds an opportunity, it is often at a lower level of abstraction than the IR exposed to the optimizer.

As originally proposed, a peephole optimizer ran as the compiler's last pass. It both read and wrote assembly code (ASM). The optimizer moved a sliding window, or "peephole," over the code. At each step, it examined the operations in the window, looking for specific patterns that it could improve. When it recognized a pattern, it would rewrite the code with a better instruction sequence. The combination of a limited pattern set and a limited window made these optimizers fast.

A classic example pattern is a store followed by a load from the same location. The load can be replaced by a copy operation.

$$\begin{array}{llcll} \texttt{storeAI} & r_1 \Rightarrow r_{arp}, 8 & & \texttt{storeAI} & r_1 \Rightarrow r_{arp}, 8 \\ \texttt{loadAI} & r_{arp}, 8 \Rightarrow r_{15} & \Longrightarrow & \texttt{i2i} & r_1 \Rightarrow r_{15} \end{array}$$

Inefficiencies similar to this one often result from statement-by-statement translation and from other optimizations, such as dead-code elimination or constant folding. Other patterns amenable to improvement by peephole optimization include simple algebraic identities, such as:

$$\begin{array}{llcll} \texttt{subI} & r_2, 0 \Rightarrow r_7 & \Longrightarrow & \texttt{mult} & r_4, r_2 \Rightarrow r_{10} \\ \texttt{mult} & r_4, r_7 \Rightarrow r_{10} & & & \end{array}$$

and cases where the target of a branch is, itself, a branch:

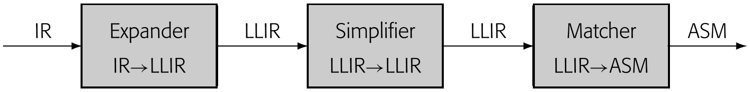
$$\begin{array}{lllcll} & \texttt{jumpI} & \rightarrow \texttt{L10} & & & \texttt{jumpI} \rightarrow \texttt{L11} \\ \texttt{L10:} & \texttt{jumpI} & \rightarrow \texttt{L11} & \Longrightarrow & \texttt{L10:} & \texttt{jumpI} \rightarrow \texttt{L11} \end{array}$$

If this eliminates the last branch to L10, then the block at L10 becomes unreachable and can be eliminated. Unfortunately, proving that the operation at L10 is unreachable takes more analysis than is typically available during peephole optimization (see Section 10.2.2).

Early peephole optimizers used a limited set of hand-coded patterns. They used exhaustive search to match the patterns but ran quickly because they used a small number of patterns and a small window size—typically two or three operations.

Peephole optimization has progressed beyond matching a small number of patterns. More complex ISAs created more opportunities that led to a more systematic approach to optimization. Modern peephole optimizers improve the code through systematic application of symbolic interpretation and simplification.

A peephole instruction selector builds on that philosophy. It breaks the selection process into three distinct tasks: expansion, simplification, and matching.



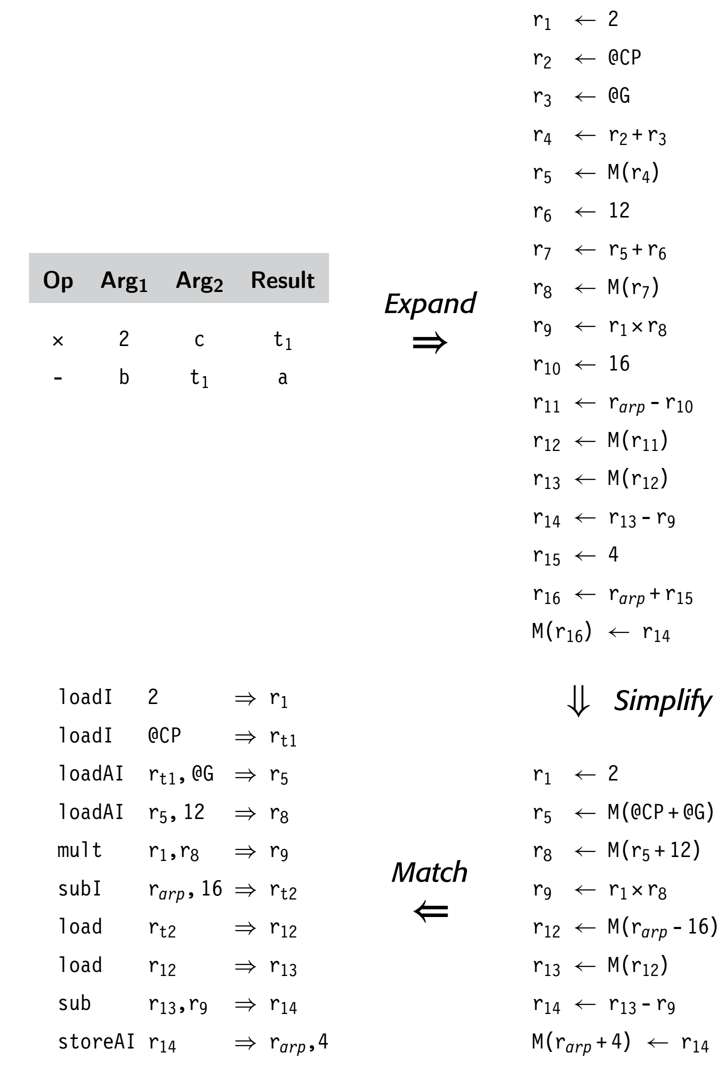
Structurally, this system looks like a compiler. It takes as input the compiler's IR; it produces as output code that resembles the target machine's assembly language (ASM). Assume that the selector processes one basic block at a time.

We carefully say "resembles" because the instruction selector may produce code that uses virtual registers rather than physical registers.

The expander rewrites the IR into a sequence of low-level IR (LLIR) operations that capture all of the block's effects—at least, all of those that affect program behavior. For example, if `add` $r_i, r_j \Rightarrow r_k$ sets the condition code, then the LLIR code must both assign $r_i + r_j$ to r_k and set the condition code to the correct value. Typically, the expander rewrites operations in a template-driven pass that ignores context.

The simplifier makes a pass over the LLIR. It slides a small window over the LLIR—say three to four operations. Within the window, it tries to improve the code by systematic application of four transformations: forward substitution, algebraic simplification, evaluation of constant-valued expressions, and elimination of dead effects, such as the creation of unused condition codes. It subjects all the details in the LLIR code to a uniform level of local optimization.

The matcher compares the simplified LLIR against a pattern library, looking for the pattern that best captures all the effects in the LLIR. The final code sequence may produce effects beyond those required by the LLIR sequence; for example, it might create new, albeit dead, condition-code values. It must, however, preserve the effects needed for correctness. It cannot eliminate a live value, regardless of whether the value is stored in memory, in a register, or in an implicitly set location such as the condition code.



Op	Arg_1	Arg_2	Result
×	2	c	t_1
-	b	t_1	a

$r_1 \leftarrow 2$
$r_2 \leftarrow \texttt{@CP}$
$r_3 \leftarrow \texttt{@G}$
$r_4 \leftarrow r_2 + r_3$
$r_5 \leftarrow \texttt{M}(r_4)$
$r_6 \leftarrow 12$
$r_7 \leftarrow r_5 + r_6$
$r_8 \leftarrow \texttt{M}(r_7)$
$r_9 \leftarrow r_1 \times r_8$
$r_{10} \leftarrow 16$
$r_{11} \leftarrow r_{arp} - r_{10}$
$r_{12} \leftarrow \texttt{M}(r_{11})$
$r_{13} \leftarrow \texttt{M}(r_{12})$
$r_{14} \leftarrow r_{13} - r_9$
$r_{15} \leftarrow 4$
$r_{16} \leftarrow r_{arp} + r_{15}$
$\texttt{M}(r_{16}) \leftarrow r_{14}$

$r_1 \leftarrow 2$
$r_5 \leftarrow \texttt{M}(\texttt{@CP} + \texttt{@G})$
$r_8 \leftarrow \texttt{M}(r_5 + 12)$
$r_9 \leftarrow r_1 \times r_8$
$r_{12} \leftarrow \texttt{M}(r_{arp} - 16)$
$r_{13} \leftarrow \texttt{M}(r_{12})$
$r_{14} \leftarrow r_{13} - r_9$
$\texttt{M}(r_{arp} + 4) \leftarrow r_{14}$

loadI 2 ⇒ r_1
loadI @CP ⇒ r_{t1}
loadAI r_{t1}, @G ⇒ r_5
loadAI r_5, 12 ⇒ r_8
mult r_1, r_8 ⇒ r_9
subI r_{arp}, 16 ⇒ r_{t2}
load r_{t2} ⇒ r_{12}
load r_{12} ⇒ r_{13}
sub r_{13}, r_9 ⇒ r_{14}
storeAI r_{14} ⇒ r_{arp}, 4

■ **FIGURE 11.2** *Expand*, *Simplify*, and *Match* Applied to the Example.

Recall that a is at ARP + 4, b is at ARP − 16, and c is at offset 12 from @G.

Fig. 11.2 summarizes how this approach might work on the example from Fig. 11.1(a). It begins, in the upper left, with the code expressed as quadruples. They compute a ← b - 2 × c. The expander creates the LLIR program shown in the upper right corner. The simplifier reduces this code to produce the LLIR code in the bottom right corner. From this LLIR program, the matcher builds the ILOC program shown in the lower left. This final program uses ten operations to implement a ← b - 2 × c.

$r_1 \leftarrow 2$
$r_2 \leftarrow @CP$
$r_3 \leftarrow @G$

Sequence 1

$r_2 \leftarrow @CP$
$r_3 \leftarrow @G$
$r_4 \leftarrow r_2 + r_3$

Sequence 2

$r_4 \leftarrow @CP + @G$
$r_5 \leftarrow M(r_4)$
$r_6 \leftarrow 12$

Sequence 3

$r_5 \leftarrow M(@CP + @G)$
$r_6 \leftarrow 12$
$r_7 \leftarrow r_5 + r_6$

Sequence 4

$r_5 \leftarrow M(@CP + @G)$
$r_7 \leftarrow r_5 + 12$
$r_8 \leftarrow M(r_7)$

Sequence 5

$r_5 \leftarrow M(@CP + @G)$
$r_8 \leftarrow M(r_5 + 12)$
$r_9 \leftarrow r_1 \times r_8$

Sequence 6

$r_8 \leftarrow M(r_5 + 12)$
$r_9 \leftarrow r_1 \times r_8$
$r_{10} \leftarrow 16$

Sequence 7

$r_9 \leftarrow r_1 \times r_8$
$r_{10} \leftarrow 16$
$r_{11} \leftarrow r_{arp} - r_{10}$

Sequence 8

$r_9 \leftarrow r_1 \times r_8$
$r_{11} \leftarrow r_{arp} - 16$
$r_{12} \leftarrow M(r_{11})$

Sequence 9

$r_9 \leftarrow r_1 \times r_8$
$r_{12} \leftarrow M(r_{arp} - 16)$
$r_{13} \leftarrow M(r_{12})$

Sequence 10

$r_{12} \leftarrow M(r_{arp} - 16)$
$r_{13} \leftarrow M(r_{12})$
$r_{14} \leftarrow r_{13} - r_9$

Sequence 11

$r_{13} \leftarrow M(r_{12})$
$r_{14} \leftarrow r_{13} - r_9$
$r_{15} \leftarrow 4$

Sequence 12

$r_{14} \leftarrow r_{13} - r_9$
$r_{15} \leftarrow 4$
$r_{16} \leftarrow r_{arp} + r_{15}$

Sequence 13

$r_{14} \leftarrow r_{13} - r_9$
$r_{16} \leftarrow r_{arp} + 4$
$M(r_{16}) \leftarrow r_{14}$

Sequence 14

$r_{14} \leftarrow r_{13} - r_9$
$M(r_{arp} + 4) \leftarrow r_{14}$

Sequence 15

$M(r_{arp} + 4) \leftarrow r_{14}$

Sequence 16

■ **FIGURE 11.3** Sequences Produced by the Simplifier.

11.3.2 **The Simplifier**

The simplifier forms the heart of the peephole process. To begin, it fills the window from the LLIR stream. Next, it tries to forward substitute defined values into uses later in the window. It applies algebraic simplification and constant folding. If any of these transformations make one of the operations in the window dead, it eliminates the dead operation and pulls a new one into the bottom of the window.

When the simplifier cannot further improve the code in the window, it emits the first operation in the window and pulls a new operation into the bottom of the window. This action "slides" the window down the block, one operation at a time.

Fig. 11.3 shows the contents of the simplifier's window at each step of the example. It assumes a three-operation window. Sequence 1 shows the window with the first three operations. No simplification is possible. The simplifier emits $r_1 \leftarrow 2$ and discards it. Next, it moves the fourth operation, which defines r_4, into the bottom of the window to create sequence 2.

1	r_1	← 2
6	r_5	← M(@CP + @G)
7	r_8	← M(r_5 + 12)
10	r_9	← $r_1 \times r_8$
11	r_{12}	← M(r_{arp} - 16)
12	r_{13}	← M(r_{12})
15	r_{14}	← r_{13} - r_9
16	M(r_{arp} + 4)	← r_{14}

Code Emitted by the Simplifier

In sequence 2, the simplifier can forward substitute both r_2 and r_3 into the addition, which makes the definitions of r_2 and r_3 both dead. The simplifier discards them and rolls the next two operations into the window. In sequence 3, it folds the definition of r_4 into the load, discards the definition of r_4, and rolls the next operation into the window. The process continues in this manner. The table in the margin shows the operations that the simplifier emits, along with the number of the sequence after which the operation is emitted. The final code shown in Fig. 11.2 consists of precisely these emitted operations.

Several issues affect the peephole selector's ability to improve code. The ability to detect when a value is dead plays a critical role in simplification. The size of the peephole window limits the optimizer's ability to combine related operations. For example, a larger window would let the simplifier fold the constant 2 into the multiply operation. The handling of control-flow operations determines what happens at block boundaries.

Recognizing Dead Values

r_{12} ← 2
r_{14} ← r_{12} + r_{12}

When the simplifier confronts a sequence similar to the one shown in the margin, it can rewrite the uses of r_{12} in the addition with 2 and evaluate the addition. It cannot, however, eliminate the definition of r_{12} unless it knows that r_{12} is dead after its uses in the add operation. Thus, the ability to recognize when a value is dead plays a critical role in the simplifier's operation.

Live analysis is discussed in Sections 8.6.1 and 9.2.2.

The compiler can compute LIVEOUT sets for each block and then, in a backward pass over the block, track which values are live at each operation. As an alternative, it can use the insight that underlies the semipruned SSA form; it can identify names that are used in more than one block and consider any such name live on exit from each block. This latter strategy avoids the expense of live analysis; it will correctly identify any value that is strictly local to the block where it is defined. All of the effects introduced by the expander are strictly local, so this cheaper approach should work well.

This analysis should be interleaved with the expansion so that it produces its results for the LLIR code.

Given either LIVEOUT sets or the set of global names, the expander can mark last uses in the LLIR. Two observations make this possible. First, the expander can process a block from bottom to top; the expansion is a simple template-driven process. Second, as it walks the block from bottom to top, the expander can build a set of values that are live at each operation, LIVENOW.

The computation of LIVENOW is simple. The expander sets the initial value for LIVENOW equal to either the LIVEOUT set for the block or the set of global names. The algorithm iterates over the operations, from the bottom of the block to its top. At each operation, r_i ← r_j op r_k, it deletes r_i

from LiveNow and adds $\texttt{r}_\texttt{j}$ and $\texttt{r}_\texttt{k}$. This algorithm produces, at each step, a LiveNow set that is as precise as the initial information used at the bottom of the block.

On a machine that uses a condition code to control conditional branches, many operations set the condition code's value. In a typical block, many of those condition code values are dead. The expander must insert explicit assignments to the condition code. The simplifier must understand when the condition code's value is dead because extraneous assignments to the condition code may prevent the matcher from generating some instruction sequences.

For example, consider the computation $\texttt{r}_\texttt{i} \times \texttt{r}_\texttt{j} + \texttt{r}_\texttt{k}$. If both × and + set the condition code, the two-operation sequence might generate the LLIR shown in the margin. The first assignment to cc is dead. If the simplifier eliminates that assignment, it can combine the other operations into a multiply-add operation, assuming that the ISA includes such an instruction. If it cannot eliminate $\texttt{cc} \leftarrow \texttt{f}_\times(\texttt{r}_\texttt{i}, \texttt{r}_\texttt{j})$, however, the matcher will not use multiply-add because it does not set cc twice.

$$
\begin{aligned}
\texttt{r}_{\texttt{t1}} &\leftarrow \texttt{r}_\texttt{i} \times \texttt{r}_\texttt{j} \\
\texttt{cc} &\leftarrow \texttt{f}_\times(\texttt{r}_\texttt{i}, \texttt{r}_\texttt{j}) \\
\texttt{r}_{\texttt{t2}} &\leftarrow \texttt{r}_{\texttt{t1}} + \texttt{r}_\texttt{k} \\
\texttt{cc} &\leftarrow \texttt{f}_+(\texttt{r}_{\texttt{t1}}, \texttt{r}_\texttt{k})
\end{aligned}
$$

LLIR for $\texttt{r}_\texttt{i} \times \texttt{r}_\texttt{j} + \texttt{r}_\texttt{k}$

Physical Versus Logical Windows

The discussion, so far, has focused on a window containing adjacent operations in the LLIR. This notion has a nice physical intuition and makes the concept concrete. However, adjacent operations in the LLIR may not operate on the same values. In fact, as target machines offer more instruction-level parallelism, a compiler's front end and optimizer must generate IR programs that have more independent and interleaved computations to keep the target machine's functional units busy. In this case, the peephole selector may find very few opportunities for improving the code.

To improve this situation, the peephole selector can use a logical window rather than a physical window. With a logical window, it considers operations that are connected by the flow of values within the code—that is, it considers together operations that define and use the same value. This creates the opportunity to combine and simplify related operations, even if they are not adjacent in the code.

During expansion, the optimizer can link each definition with the next use of its value in the block. The simplifier uses these links to fill its window. When the simplifier reaches operation *i*, it constructs a window for *i* by pulling in operations linked to *i*'s result. Using a logical window within a block can make the simplifier more effective, reducing both the compile time required and the number of operations remaining after simplification.

In the example, the simplifier cannot fold the constant two into the multiply operation because the two operations are never in the window together. In a simplifier with a logical window, the multiply and the immediate load would be in the window together, so the simplifier could substitute the value two into the multiply, and eliminate the immediate load operation.

We can extend this idea to multiple blocks. The compiler can attempt to simplify operations that are logically adjacent, even if they are in different blocks. Using multiple blocks requires a global analysis to determine which uses each definition can reach, such as reaching definitions from Section 9.2.4. Additionally, the simplifier must recognize that one definition may reach multiple uses, and one use might refer to values computed by several distinct definitions. Thus, the simplifier cannot simply combine the defining operation with one use and leave the remaining operations stranded. It must either limit its consideration to simple situations, such as a single definition and a single use or multiple uses with a single definition, or it must perform detailed analysis to determine whether a combination is both safe and profitable. Restricting the logical window to an extended basic block avoids some of these complications.

11.3.3 The Matcher

The final component of a peephole selector is the matching phase that maps an arbitrary LLIR sequence onto the target machine's ISA. Peephole selectors have used different technologies to implement the matcher, ranging from parsers adapted to the task to ad-hoc code. The choice between these approaches depends heavily on the size and complexity of the target instruction set and the availability of tools.

The goal for a generated instruction selector is to reduce the amount of work needed to retarget a compiler. With a specification-driven matcher, the compiler writer creates a description of the ISA and its relationship to the LLIR; the build-time generator then constructs the matcher that runs at compile time. The compiler writer must ensure that (1) the description covers the full set of possible LLIR sequences and (2) the description makes good use of the target ISA, including its address modes and specialized operations.

With a hand-coded matcher, the compiler writer modifies the matcher directly so that it emits the appropriate target ISA operations. That task sounds daunting; however, the differences between processor ISAs are typically smaller than the similarities between them. Logical and arithmetic operations function in similar ways across most architectures; the differences lie in the way they name operands and their side effects on system state (such as setting a condition code).

Several important compiler systems have used peephole selectors. The Gnu compiler system, GCC, uses an LLIR called register-transfer language (RTL) for some optimizations and for code generation. The back end uses a peephole selector to convert RTL into target-machine ASM. The matcher interprets the RTL as trees and uses a tree-pattern matcher built from a target-machine description. By contrast, the matcher in Davidson's VPO uses a parser that treats the LLIR as a linear form and emits target-machine code as it parses. It builds the selector from a description of the target ISA and from its own knowledge of the LLIR used in the system.

The parser must handle the ambiguities introduced by the presence of multiple ways to implement a given LLIR sequence.

These peephole-based systems have two fundamental strengths.

- Systematic late-stage simplification lets the compiler eliminate inefficiencies left behind by earlier phases in the translation.
- Forward substitution and constant folding expose opportunities to map LLIR sequences into processor address modes.

The net result is a final program that executes fewer operations and takes better advantage of the target machine's instruction set.

SECTION REVIEW

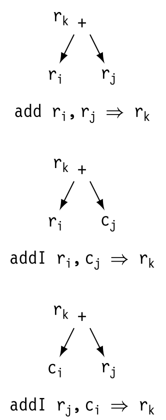
Peephole optimization can form the basis of a fast and effective instruction selector. A peephole selector consists of a template-driven expander that translates the compiler's IR into a more detailed low-level IR; a simplifier that uses forward substitution, algebraic simplification, constant propagation, and dead-code elimination within a small, moving window; and a matcher that maps the optimized low-level IR onto the target ISA.

The strength of this approach lies in the simplifier; it removes interoperation inefficiencies introduced in the translation process. It focuses on opportunities that involve local values; many of those opportunities are hidden at earlier stages of translation. The resulting improvements can be surprising. The final matching phase is straightforward; technologies ranging from hand-coded matchers to LR parsers have been used.

REVIEW QUESTIONS

1. Sketch an algorithm for the simplifier. What is the algorithm's complexity? How does peephole-window size affect the time bound?
2. How might you extend the simplifier so that it uses the algebraic identities from Fig. 8.3 to improve the LLIR code?

11.4 SELECTION VIA TREE-PATTERN MATCHING


r_k +
r_i r_j
add $r_i, r_j \Rightarrow r_k$
r_k +
r_i c_j
addI $r_i, c_j \Rightarrow r_k$
r_k +
c_i r_j
addI $r_j, c_i \Rightarrow r_k$

Operation Trees for add and addI

Another way to attack the complexity of instruction selection is with tree-pattern matching tools. To transform code generation into a tree-pattern matching problem, both the IR form of the program and the target machine's instruction set must be expressed as trees. The compiler can use a low-level AST as a detailed model of the code being compiled. It can use similar trees to represent the operations available on the target processor. For example, ILOC's addition operations might be modeled by operation trees like those shown in the margin for add and addI. By systematically matching such operation trees, or pattern trees, with subtrees of an AST, the compiler can discover all the potential implementations for the subtree.

Given a low-level AST for the code and a collection of operation trees for the target ISA, the matcher constructs a *tiling* of the AST with operation trees. A tile is a pair, $\langle x,y \rangle$, where x is a node in the AST and y is the root of an operation tree. A tiling is a set of pairs that meet the following constraints:

1. The set of tiles covers every AST node.
2. The root of each operation tree overlays a leaf in another operation tree, unless it overlays the root of the AST.
3. Where two operation trees overlap, they are *compatible*; that is, they agree in both storage class and value type.
4. The overlap between any two operation trees in the set occurs at a single node.

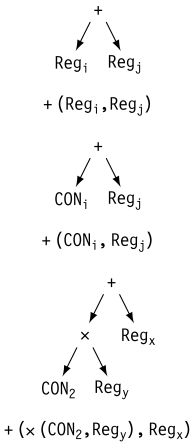
A set of pairs that meets these criteria is a tiling; it *implements* the AST.

The presence of a pair $\langle x, y \rangle$ in the tiling indicates that the AST subtree rooted at x can be implemented by the operation tree rooted at y. Unless x is a leaf, the choice of y will depend on finding implementations for subtrees of x that work with y. To build a tiling, the compiler must ensure that both the entire tree and each of its subtrees can be implemented by the specified set of operation trees.

Given a tiling that implements an AST, the compiler can generate assembly code in a bottom-up walk over the AST. Thus, the key to making practical instruction selectors based on tree-pattern matching lies in algorithms that find good tilings for an AST. Tools exist for several different techniques: tree matching, text matching, and bottom-up rewrite systems (BURSs). The tools associate costs with operation trees and produce minimal-cost tilings. This section focuses on tree matching, but the insights carry over to text matching and BURSs.

11.4.1 Representing Trees

We need a notation to describe both ASTs and tree patterns. Prefix notation is well-suited to this task. For example, the add operation shown in the margin is $+(\mathtt{Reg}_i, \mathtt{Reg}_j)$ in prefix form, and the addI operation is $+(\mathtt{CON}_i, \mathtt{Reg}_j)$. This same notation works for both low-level ASTs that represent executable code and the pattern trees that represent target-machine operations. The examples in this chapter limit themselves to integer operations. Extending the rules to other data types adds many new patterns, but few new insights.



$+(\mathtt{Reg}_i,\mathtt{Reg}_j)$

$+(\mathtt{CON}_i, \mathtt{Reg}_j)$

$+(\times(\mathtt{CON}_2,\mathtt{Reg}_y), \mathtt{Reg}_x)$

LLIR Trees and Their Prefix Descriptions

The operands of a subtree are either subtrees or leaves. Subtrees are expressed in prefix notation, as with the multiply subtree in the AST shown in the margin: $+(\times(\mathtt{CON}_2, \mathtt{Reg}_y), \mathtt{Reg}_x)$. Leaves are assigned symbolic names that encode information about the type and storage location of the operand. For example, $\mathtt{Reg}_i$ indicates a value that resides in a register and $\mathtt{CON}_j$ indicates a constant value. In an AST, labels such as VAL, NUM, LAB, and CON provide more detailed information (see page 585). Subscripts are added to names for uniqueness. We can rewrite the AST from Fig. 11.1(b) into prefix form:

$$
\begin{aligned}
\leftarrow\ \ &(+(\mathtt{VAL}_1, \mathtt{NUM}_1),\\
&\ -(\blacklozenge(\blacklozenge(-(\mathtt{VAL}_2, \mathtt{NUM}_2))), \times(\mathtt{NUM}_3, \blacklozenge(+(\mathtt{LAB}_1, \mathtt{NUM}_4)))))
\end{aligned}
$$

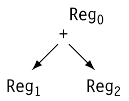
While the drawing of the tree may be more intuitive, this linear prefix form contains precisely the same information.

Throughout this section, we will assume that the IR uses virtual names; that is, the supply of names is unlimited (see Section 1.4). After selection, a register allocator will map the virtual names onto the target machine's limited set of physical registers (see Chapter 13).

11.4.2 Rewrite Rules

To build an instruction selector based on tree-pattern matching, the compiler writer creates a set of rewrite rules that maps the AST into the target machine's ISA. Each rule consists of a tree pattern, a code template, and an associated cost. These tree patterns describe the structure of the AST. Fig. 11.4 shows the ILOC subset that we will use in the ongoing examples.

The patterns resemble productions in a context-free grammar (CFG). Each pattern has a left-hand side (LHS) and a right-hand side (RHS); as in a CFG, the LHS and RHS are separated by the derives symbol, →. The drawing in the margin shows the tree for a three-register add and its corresponding pattern. The labels give names to the various definitions and uses. The input arguments are $\mathtt{Reg}_1$ and $\mathtt{Reg}_2$, respectively. The output of the operation is $\mathtt{Reg}_0$.



$\mathtt{Reg}_0 \rightarrow +(\mathtt{Reg}_1,\mathtt{Reg}_2)$

LLIR Tree and Its Pattern

Arithmetic Operations				Memory Operations			
add	r_1, r_2	⇒	r_3	store	r_1	⇒	r_2
addI	r_1, c_2	⇒	r_3	storeAO	r_1	⇒	r_2, r_3
sub	r_1, r_2	⇒	r_3	storeAI	r_1	⇒	r_2, c_3
subI	r_1, c_2	⇒	r_3	loadI	c_1	⇒	r_3
rsubI	r_2, c_1	⇒	r_3	load	r_1	⇒	r_3
mult	r_1, r_2	⇒	r_3	loadAO	r_1, r_2	⇒	r_3
multI	r_1, c_2	⇒	r_3	loadAI	r_1, c_2	⇒	r_3

■ **FIGURE 11.4** The ILOC Subset.

Some symbols appear exclusively on the RHS of patterns. These symbols are analogous to the terminal symbols in a CFG; they represent concrete symbols that can be leaves in a tree. Other symbols appear on either the LHS or RHS of a pattern. These symbols are analogous to the nonterminal symbols in a CFG; they are syntactic variables used to tie productions together into sequences.

A rule set consists of a collection of related rules that, together, can tile a tree. Fig. 11.5 shows a rule set for the low-level AST that we described in Section 11.2.2. Each rule represents a small AST. For example, rule 8 was shown earlier in the margin. Its RHS describes the + node and its children. The rule's LHS corresponds to the label on the + node, $\texttt{Reg}_0$ in the margin drawing.

Rule 19 is too simple. In practice, the compiler might place the constant pool at a known offset from the procedure's entry point. The prolog code could either load @CP to a register or store it in the local data area.

Rule 19 deals with a common problem: the code needs to load a constant value that is too large to accommodate in a load immediate operation. The code template suggests one way to solve the problem. It assumes that each procedure has a unique, statically initialized constant pool, and it represents the start of that constant pool with the symbol @CP. Further, it assumes that @L is the positive offset of $\texttt{LAB}_1$ from @CP. With these assumptions, the emitted code loads @CP into a register and uses it as the base address for a loadAI operation.

The rules in Fig. 11.5 describe the set of potential ASTs for a list of assignments. Not all the rules produce code. For example, rules 0 and 1 create an ordered list of Stmt, the nonterminal symbol for an assignment statement. Rules 20 through 23 handle address mode selection and rule 24 is used in immediate-mode operations.

Interactions between the patterns, encoded through the use of LHS symbols, define the ways in which subtrees can combine. For example, productions 5 through 20 each have Reg on their LHS. Thus, each of those rules describes a subtree that can rewrite a Reg.

	Production			Cost	Code Template	
0	Goal	→	Goal Stmt	0		
1	Goal	→	Stmt	0		
2	Stmt	→	←(Reg$_1$,Reg$_2$)	3	store	r$_2$ ⇒ r$_1$
3	Stmt	→	←(T1$_1$,Reg$_2$)	3	storeAO	r$_2$ ⇒ T1.r$_1$,T1.r$_2$
4	Stmt	→	←(T2$_1$,Reg$_2$)	3	storeAI	r$_2$ ⇒ T2.r,T2.n
5	Reg	→	◆(Reg$_1$)	3	load	r$_1$ ⇒ r$_{new}$
6	Reg	→	◆(T1$_1$)	3	loadAO	T1.r$_1$,T1.r$_2$ ⇒ r$_{new}$
7	Reg	→	◆(T2$_1$)	3	loadAI	T2.r,T2.n ⇒ r$_{new}$
8	Reg	→	+(Reg$_1$,Reg$_2$)	1	add	r$_1$,r$_2$ ⇒ r$_{new}$
9	Reg	→	+(Reg$_1$,T3$_2$)	1	addI	r$_1$,T3 ⇒ r$_{new}$
10	Reg	→	+(T3$_1$,Reg$_2$)	1	addI	r$_2$,T3 ⇒ r$_{new}$
11	Reg	→	-(Reg$_1$,Reg$_2$)	1	sub	r$_1$,r$_2$ ⇒ r$_{new}$
12	Reg	→	-(Reg$_1$,T3$_2$)	1	subI	r$_1$,T3 ⇒ r$_{new}$
13	Reg	→	-(T3$_1$,Reg$_2$)	1	rsubI	r$_2$,T3 ⇒ r$_{new}$
14	Reg	→	×(Reg$_1$,Reg$_2$)	2	mult	r$_1$,r$_2$ ⇒ r$_{new}$
15	Reg	→	×(Reg$_1$,T3$_2$)	2	multI	r$_1$,T3 ⇒ r$_{new}$
16	Reg	→	×(T3$_1$,Reg$_2$)	2	multI	r$_2$,T3 ⇒ r$_{new}$
17	Reg	→	CON$_1$	1	loadI	CON$_1$ ⇒ r$_{new}$
18	Reg	→	NUM$_1$	1	loadI	NUM$_1$ ⇒ r$_{new}$
19	Reg	→	LAB$_1$	4	loadI loadAI	@CP ⇒ r$_{new_1}$ r$_{new_1}$,@L ⇒ r$_{new_2}$
20	Reg	→	VAL$_1$	0		
21	T1	→	+(Reg$_1$,Reg$_2$)	0		
22	T2	→	+(Reg$_1$,T3$_2$)	0		
23	T2	→	+(T3$_1$,Reg$_2$)	0		
24	T3	→	NUM$_1$	0		

■ **FIGURE 11.5** Rewrite Rules for Tiling the Low-Level Tree with ILOC.

The LHS symbols encode knowledge about the type and storage class of the values that they represent. For example, a Reg represents a value in a register. A Reg might result from a subtree that produces an integer value stored in a register, as in rules 5 through 16. It might also result from a CON, a NUM, or a LAB, through productions 17, 18, and 19. Rule 20 provides a way to convert a VAL node to a Reg—essentially, it names the value. The VAL might be a global value, such as the ARP, or it might be the result of a computation performed in a disjoint subtree, such as a redundant expression found by the optimizer.

T1 and T2 represent addresses—values that can be computed in address modes such as loadAO and loadAI. T3 represents a NUM used directly as an immediate operand.

Finally, note that the rule set is ambiguous. Rules 9 and 22 have the same RHS, as do rules 10 and 23. Those pairs have different purposes and different costs. Rule 9 represents an explicit, general purpose add operation, while rule 22 represents an add operation done by the memory address hardware.

Cost Estimates

Some systems only allow fixed costs. Others let the costs vary during matching to reflect prior choices.

Each rule has a cost; that cost should provide a realistic estimate of the runtime cost of executing the code in the template. Fig. 11.5 assumes that addition and subtraction require one cycle, multiplication takes two, and memory operations need three cycles, a reasonable estimate for a hit in the first-level data cache. Rules that generate no code, such as rule 22, have zero cost. The cost estimates will drive choices among alternative rules.

A full rule set might have multiple rules that require more than one target-machine operation, as rule 19 does. The tree-matching algorithm sees such a rule as a single pattern. As long as the cost estimates are accurate, the presence of multiple-operation sequences should not complicate matters.

Building a Rule Set

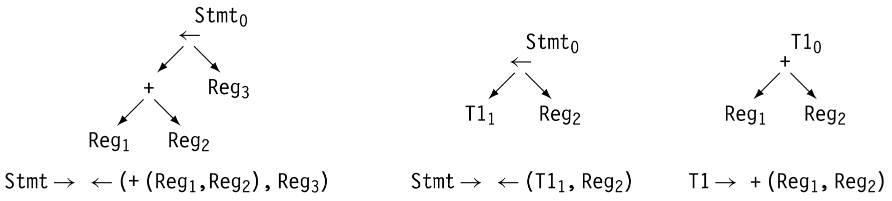
A common strategy in constructing a rule set is to begin with a minimal set of rules that covers the tree. That is, the compiler writer can derive a set of rules that has a pattern for every node in the AST and a correct code sequence for each pattern. Once that rule set has been designed, tested, and debugged, the compiler writer can add rules with patterns that use specialized operations to handle more complex cases.

Restrictions on the Rule Set

To simplify the code in the tree-pattern matcher, we restrict the rule set in two ways. First, *each pattern includes at most one operator*. Any multioperator rule can be rewritten into a sequence of single-operator rules.

This special case matters if the hardware can compute the sum in the address unit, as in storeAO.

Consider two possible rule sets for an ILOC storeAO operation, shown in Fig. 11.6. Panel (a) shows a single rule, ← (+ (Reg_1,Reg_2),Reg_3). It specializes the general store operation described by rule 2 for the case when the address is the sum of two registers. Panel (b) shows two single-operator patterns that together tile the same subtree. To create the single-operator rules, the compiler writer introduced a new name, T1, that appears only in the address computation of a storeAO or a loadAO. The rules in panel (a) and (b) are equivalent; they tile the same set of LLIR subtrees.



(a) Pattern for storeAO

(b) Single-Operator Patterns for storeAO

■ **FIGURE 11.6** Single-Operator Patterns Versus Multioperator Patterns.

The restriction to single-operator rules greatly simplifies the code for the tree-pattern matcher. To match the pattern in panel (a), the matcher must inspect both the + node and the ← node together, or it must carry along explicit state to track such matches. Either approach adds complexity and cost to the implementation.

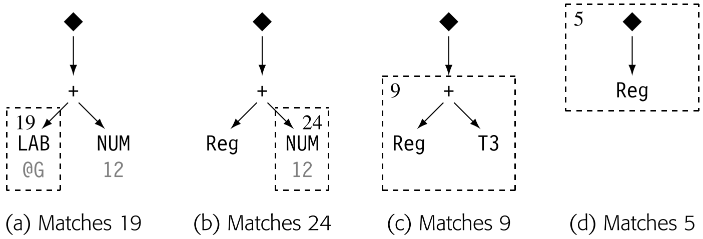
With the two single-operator patterns in panel (b), the matcher makes purely local decisions. The AST subtree + (Reg,Reg) matches both the general rule that describes an add (rule 8) and the rule that describes a T1 (rule 21). Moving up to that subtree's parent, the matcher can produce matches to each of rule 2, for the left child's label as a Reg, and rule 3, for the left child's label as a T1.

The key to making single operator rules produce the desired results for complex target-machine operations lies in the cost structure of the rule set. With single operator patterns, the sum of the costs of the single operator rules should equal the cost of the complex instruction. If the cost of the operation is one cycle and it takes two patterns to match it, then they can be assigned arbitrary nonnegative costs that sum to one, such as $\frac{1}{2}$ and $\frac{1}{2}$, or $\frac{1}{4}$ and $\frac{3}{4}$, or 0 and 1. This approach ensures that a low-cost match generates the instruction sequence with lowest local cost.

The second restriction we impose is that *leaves in the tree appear only in singleton rules*, such as rules 17 to 20 and 24. Again, the intent is to simplify the tree-pattern matcher. For the situation where a NUM needs to move into a register, rule 18, Reg ← NUM, accomplishes the task. When the NUM can appear as an immediate field, it matches rule 24, T3 ← NUM. Here, T3 is an LHS symbol that only appears in an immediate field of an operation such as addI or loadAI. If the target ISA supported multiple distinct lengths of immediate operands, the rule set would include an LHS symbol for each length.

Notice that rule 18 has a code template, but rule 24 does not.

This second restriction avoids special case code in the tree-pattern matcher. When the matcher considers the children of an operator, it can assume that



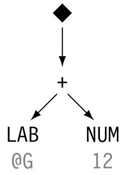
■ FIGURE 11.7 A Simple Tree Rewrite Sequence.

those children have already been annotated with type and cost information by the singleton rules. The code for operators remains simple and uniform.

Either one or both of these restrictions can be discarded, at the cost of additional complexity in the tree-pattern matcher. In a hand-coded matcher, this simplicity leads to efficient matchers. An automatically generated, table-driven scheme may have different cost tradeoffs.

11.4.3 Computing Tilings

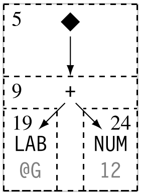
Given a rule set that maps the compiler's IR onto the target machine's ISA, the instruction selector must map a specific AST into an instruction sequence. To do this, a tree-pattern matching code generator constructs a tiling of the AST with the tree-patterns from the rule set. Several techniques for constructing such a tiling exist; they are similar in concept but vary in detail.



Example Subtree

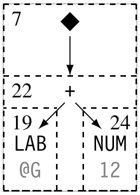
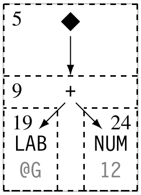
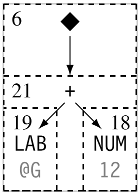
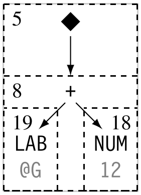
To help us understand tiling, consider the AST shown in the margin, a subtree from Fig. 11.1(b). It describes a memory operation that loads a word from the address at offset 12 from the label @G. Using the rules from Fig. 11.5, we can find a sequence of rules that tiles the tree.

Fig. 11.7 shows one way to tile this AST. The figure treats the rules as rewrites to the tree. As shown in panel (a), rule 19 matches the LAB node. Rule 19 rewrites the LAB as a Reg, shown in panel (b). Rule 24 matches the NUM node and rewrites it as a T3 to produce the AST shown in panel (c). One of the matches for +(Reg,T3) in panel (c) is rule 9, which rewrites that subtree as a Reg, as shown in panel (d). Rule 5 matches the AST in panel (d) and rewrites it as a single Reg node, not shown. We denote this rewrite sequence as ⟨19,24,9,5⟩; order in the sequence matches the order of application.



⟨19,24,9,5⟩
⟨24,19,9,5⟩

Rules 19 and 24 can be applied in either order. Thus, the sequence ⟨19,24,9,5⟩ produces the same tiling as ⟨24,19,9,5⟩. The diagram in the margin summarizes those sequences.

Tiling				
Code	`loadI @CP` ⇒ r_i `loadAI` r_i,`@G` ⇒ r_j `loadAI` r_j,`12` ⇒ r_k Cost: 7 cycles	`loadI @CP` ⇒ r_i `loadAI` r_i,`@G` ⇒ r_j `addI` r_j,`12` ⇒ r_k `load` r_k ⇒ r_l Cost: 8 cycles	`loadI @CP` ⇒ r_i `loadAI` r_i,`@G` ⇒ r_j `loadI 12` ⇒ r_k `loadAO` r_j,r_k ⇒ r_l Cost: 8 cycles	`loadI @CP` ⇒ r_i `loadAI` r_i,`@G` ⇒ r_j `loadI 12` ⇒ r_k `add` r_j,r_k ⇒ r_l `load` r_l ⇒ r_m Cost: 9 cycles
Sequences	⟨19,24,22,7⟩ ⟨24,19,22,7⟩	⟨19,24,9,5⟩ ⟨24,19,9,5⟩	⟨19,18,21,6⟩ ⟨18,19,21,6⟩	⟨19,18,8,5⟩ ⟨18,19,8,5⟩

■ **FIGURE 11.8** The Set of All Tilings for the Example Subtree.

This tiling implements the AST; it meets the four criteria defined on page 596. It covers each AST node with one or two operation-tree nodes. The root of each pattern tree overlaps with a leaf in its parent. For example, the root of 19 is a leaf in 9. The connections between patterns are compatible and limited to one node in each subtree. Where two operation-tree nodes cover an AST node, the connected nodes have the same nonterminal label. Thus, this tiling implements the AST.

Given a valid tiling, the selector generates code using the templates associated with each rule. The code template consists of zero or more operations that, together, implement the subtree covered by the rule. For example, the sequence ⟨19,24,9,5⟩ implies four operations. Rule 19 produces a `loadI` followed by a `loadAI`; rule 24 generates no code; rule 9 generates an `addI`; and rule 5 generates the `load`. The resulting code appears in the second column of Fig. 11.8, with abstract register names to create the flow of values. Note that ⟨24,19,9,5⟩ produces the same code, because rule 24 produces no code. In general, sequences that differ in the order of rule application may generate the operations in a different order.

Choosing Among Multiple Tilings

Eight different sequences implement the example AST. Fig. 11.8 shows all eight sequences; those that differ only in order appear together. The top row shows the tilings. The middle row shows the code generated by each tiling and its cost. The bottom row shows the sequences for each tiling.

The sequences that use both rules 18 and 19 would generate the two `loadI`s in different orders for ⟨18,19⟩ and ⟨19,18⟩.

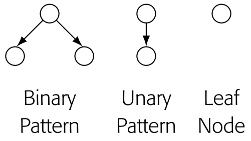
```
Tile(n)                         /* n is an AST node */
   if n is a leaf then
      Match(n,∗) ← { rules that implement n }
   else if n is a unary node then
      Tile(child(n))
      Match(n,∗) ← ∅         /* Clear n's Match sets */
      for each rule r where operator(r) = operator(n) do
         if (child(r), child(n)) are compatible then
            add r to Match(n, class(r))
   else if n is a binary node then
      Tile(left(n))
      Tile(right(n))
      Match(n,∗) ← ∅         /* Clear n's Match sets */
      for each rule r where operator(r) = operator(n) do
         if (left(r), left(n)) and (right(r), right(n)) are compatible then
               add r to Match(n, class(r))
```

■ **FIGURE 11.9** Compute All Matches to Tile an AST.

To emit code, the instruction selector must choose one sequence. The obvious choice is to take the low-cost sequence. If the cost estimates for the rules reflect actual runtime costs, then the low-cost sequence should be fastest. Note that the costs can reflect properties other than execution speed. For example, in an application where code space is critical, a cost metric that reflects the byte-length of the code sequence might be appropriate.

A Tiling Algorithm

Fig. 11.9 shows a simple recursive algorithm that computes the set of matches for each node in the tree. To keep the exposition simple, the algorithm assumes that the AST and the rule set consist of nodes with either zero, one, or two operands.



This restriction allows the algorithm to deal with just three kinds of tree-patterns: a binary node and its two children, a unary node and its child, or a leaf node. This limitation keeps the matcher small and efficient.

Tile annotates each node with a vector of rules that match the subtree rooted at that node. The vector has one element for each LHS symbol in the rule set. By design, each LHS symbol in the rule set corresponds to a ⟨storage class,type⟩ pair. For a node n, the row *Match(n,∗)* contains all of the rules that can implement the subtree rooted at n. *Match(n,c)* shows all of the rules that can implement the subtree rooted at n, to produce a result in c.

For a leaf node, the set of matches can be precomputed. In our example rule set, a NUM node always has the *Match* set shown in the margin. The rules provide one way to rewrite NUM as a Reg, with rule 18, and one way to rewrite a NUM as a T3, with rule 24. Each leaf node has its own vector.

Reg	Stmt	T1	T2	T3
18				24

Match Set for NUM Node

For a unary node *n*, *Tile* first recurs on *n*'s child to discover the ways it can be implemented. It then examines each rule *r* that implements *n*'s operator. If the child has an implementation that is compatible with *r* marked in its *Match* set, then *Tile* adds *r* to *Match(n, class(r))*, where *class(r)* denotes the LHS symbol of rule *r*.

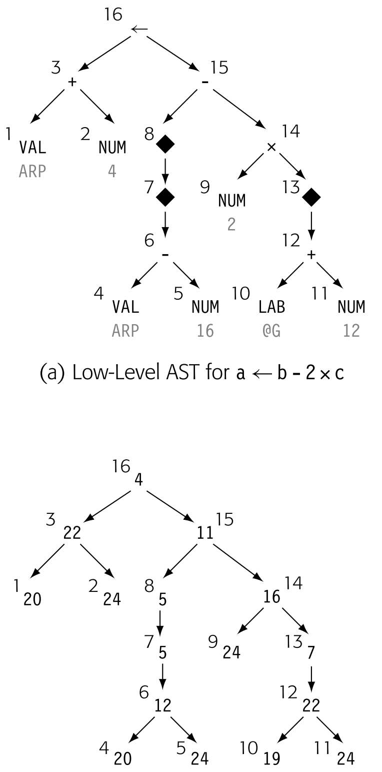
To test compatibility between rule *r* and the implementations for AST node *n*'s child, the algorithm uses a simple test based on the definition of compatibility: the rule and the AST must agree on the operator and on the storage class and value type of the child. Given an AST subtree $o_1(x)$ and a tree pattern $o_2(y)$, the two are compatible if and only if (1) the operators in o_1 and o_2 are the same; and (2) *Match(x, class(y))* is nonempty. The first condition ensures that the tree pattern and the AST subtree use the same operation. The second ensures that the child in the AST can be implemented with a rule that produces the kind of value that rule *r* uses.

For a binary node *n*, *Tile* follows the same plan as for a unary node. It recurs on both the left and right children in the AST. It tests compatibility between the left child in the AST and the corresponding term in rule *r* and the right child in the AST and the corresponding term in *r*. If *r* uses the same operator as *n* and *r* is compatible with the tilings computed for *n*'s children, then *Tile* adds *r* to *Match(n, class(r))*.

A Worked Example

Fig. 11.10 shows the results of applying *Tile* to our continuing example. Panel (a) shows the AST for a ← b - 2 × c. Nodes in the AST are annotated with their postorder numbers; *Tile* traverses the tree in postorder. Panel (d) shows all of the matches that *Tile* finds for each node. Superscripts on the rule numbers indicate the lowest cost for each entry. The low-cost entry in the row is set in **bold** text.

To gain a feel for the algorithm, consider the subtree that starts with node 13, which is the example from Fig. 11.8. In postorder, *Tile* will compute the *Match* sets for nodes 10 and 11. As leaves, these are precomputed sets. The only rule that applies to a LAB is rule 19, which rewrites LAB with Reg. The table entry reflects that fact. Similarly, the entry for node 11 is the *Match* set shown earlier, which indicates that rule 18 rewrites a NUM into a Reg, and rule 24 rewrites a Num into a T3.



(a) Low-Level AST for a ← b - 2 × c

(b) Top-down Assignment of Rules

Node	Rule	Emitted Code			
6	12	subI	r_{arp}, 16	⇒	r_a
7	5	load	r_a	⇒	r_b
8	5	load	r_b	⇒	r_c
10	19	loadI	@CP	⇒	r_d
		loadAI	r_d, @G	⇒	r_e
13	7	loadAI	r_e, 12	⇒	r_f
14	16	multI	r_f, 2	⇒	r_g
15	11	sub	r_c, r_g	⇒	r_h
16	4	storeAI	r_h	⇒	r_{arp}, 4

(c) Code Emitted by Rules

Node	Reg	Stmt	T1	T2	T3
1	**20^{0}**				
2	18^{1}				**24^{0}**
3	8^{2} 9^{1}		21^{1}	**22^{0}**	
4	**20^{0}**				
5	18^{1}				**24^{0}**
6	11^{2} **12^{1}**				
7	**5^{4}**				
8	**5^{7}**				
9	18^{1}				**24^{0}**
10	**19^{4}**				
11	18^{1}				**24^{0}**
12	8^{6} 9^{5}		21^{5}	**22^{4}**	
13	5^{8} 6^{8} **7^{7}**				
14	14^{10} **16^{9}**				
15	**11^{17}**				
16		2^{21} 3^{21} **4^{20}**			

(d) Full Set of Matches and Costs

■ **FIGURE 11.10** Results of Running *Tile* on the Low-Level AST for a ←b - 2 × c.

For node 12, *Tile* finds four possibilities. It can produce a Reg for the addition, using either rule 8 or rule 9. It can produce a T1 using rule 21, or it can produce a T2 using rule 22. As shown in Fig. 11.8, each of those choices dictates specific matches for nodes 10 and 11.

Finally, for node 13, *Tile* finds three choices: rules 5, 6, and 7. Each of these rewrites the subtree with a Reg. A top-down walk in the AST subtree, choosing a compatible set of rules from *Match* will generate the sequences shown in Fig. 11.8.

Accounting for Costs

Given the set of matches and the costs for each rule, the compiler can compute the least cost choice in each category, during a simple postorder traversal. It accumulates costs for each category, bottom up, by determining the cost for a specific rule choice as the rule's own cost, plus the cost of the choices at subtrees that the rule requires. The matcher can compute the cost for each rule choice and keep the smallest one for each category.

The cost computation can be folded into *Tile*, as shown in Fig. 11.11. The code is a straightforward extension of Fig. 11.9.

The table in Fig. 11.10(d) shows where, during the matching process, choices occur. Consider node 3. Two patterns match the subtree to produce a value of type Reg. Rule 8 has a total cost of 2: 1 for itself, plus 0 for a Reg at node 1 and 1 for a Reg at node 2. Rule 9 has a total cost of 1: 1 for itself, plus 0 for a Reg at node 1 and 0 for a T3 at node 2. The cost-driven algorithm will keep the match to rule 9. In addition to these matches for Reg, one pattern matches the subtree to produce each of T1 and T2. The low-cost among these matches for node 3 is rule 22, which produces a T2.

By the time the matcher finishes, it has annotated each node with the rule that produces the low-cost match. These matches are shown in Fig. 11.10(b). The instruction selector can then emit code in a bottom-up, postorder pass over the tree. Fig. 11.10(c) shows the resulting code, with appropriate register names used to tie together the various code templates.

This process yields, at each point, a rule choice that produces a minimal cost code sequence in some local neighborhood in the tree. The literature refers to this property as *local optimality*.

Local optimality
A scheme in which the compiler has no better alternative, at each point in the code, is considered *locally optimal*.

Local optimality does not guarantee that the solution is optimal from a broader perspective, such as the entire procedure or the entire program. In general, compilers cannot efficiently consider all the details that would be necessary to achieve global optimality.

```
Tile(n)    /* n is a node in an AST */
  if n is a leaf node then
    Match(n,*).rule ← {low-cost rule that matches n, in each class}
    Match(n,*).cost ← {corresponding cost}
  else if n is a unary node then
    Tile(child(n))
    Match(n,*).rule ← invalid
    Match(n,*).cost ← largest integer
    for each rule r where operator(r) = operator(n) do
      if (child(r), child(n)) are compatible then
        NewCost ← RuleCost(r) + Match(child(n),class(child(r))).cost
        if (Match(n,class(r)).cost > NewCost) then
          Match(n,class(r)).rule ← r
          Match(n,class(r)).cost ← NewCost
  else if n is a binary node then
    Tile(left(n))
    Tile(right(n))
    Match(n,*).rule ← invalid
    Match(n,*).cost ← largest integer
    for each rule r where operator(r) = operator(n) do
      if (left(r), left(n)) and (right(r), right(n)) are compatible then
        NewCost ← RuleCost(r) + Match(left(n), class(left(r))).cost
                     + Match(right(n), class(right(r))).cost
        if (Match(n,class(r)).cost > NewCost) then
          Match(n,class(r)).rule ← r
          Match(n,class(r)).cost ← NewCost
```

■ **FIGURE 11.11** Compute Low-Cost Matches to Tile an AST.

Notice how different the final code is for the three variable references. The store to a, at offset 4 from the ARP, folds completely into a single storeAI operation. The load from b, a call-by-reference parameter whose pointer is 16 bytes before the ARP requires a subI and two load operations; the address-offset address mode only handles positive offsets, and the call-by-reference binding leads to an extra load. Finally, the load from c, a global variable stored at offset 12 from a label, requires a loadI to find the constant pool, followed by two loadAI operations. The first loadAI fetches the address of the label stored at @CP + @G, and the second fetches the value at offset 12 from that address. At the source level, all three references look textually similar; the code generated for them is not.

TREE-PATTERN MATCHING ON QUADS?

The terms used to describe these techniques—*tree-pattern matching* and *peephole optimization*—contain implicit assumptions about the kinds of IR to which they can be applied. The description of tree-pattern matching selectors implies that the selector operates on a tree-shaped IR. Similarly, peephole optimizers were first proposed as a final assembly-to-assembly improvement pass. The idea of a moving instruction window strongly suggests a linear, low-level IR for a peephole selector.

Both techniques can be adapted to fit most IRs. A compiler can interpret a low-level linear IR like ILOC as trees. Each operation becomes a tree node; the edges are implied by the reuse of operands. Similarly, if the compiler assigns a name to each node, it can interpret trees as a linear form by performing a postorder treewalk. A clever implementor can adapt the methods presented in this chapter to a wide variety of actual IRs.

As a final point, notice that the tree-pattern matching selector folds the constant two into the multiply operation. Recall that the peephole selector was unable to perform that rewrite because the assignment of two to r_1 and the use of r_1 in the multiply were too far apart for the simplifier to see the opportunity. By contrast, the tree-pattern matcher encodes the fact that the subtree has a constant value and carries it along in the set of possible matches to the point where the decision between a `mult` and a `multI` occurs.

11.4.4 **Tools**

As we have seen, a tree-oriented, bottom-up approach can produce efficient instruction selectors. There are several ways that the compiler writer can implement code generators based on these principles.

1. The compiler writer can hand code a matcher, similar to *Tile*, that explicitly checks for matching rules as it tiles the tree. A careful implementation can limit the set of rules examined at each node. This avoids the large table and leads to a compact code generator.
2. Since the problem is finite, the compiler writer can encode it as a finite tree-matching automaton, similar to a DFA. In this scheme, the lookup table encodes the automaton's transition function and implicitly incorporates all the required state information. Multiple systems have used this approach, often called a bottom-up rewrite system (BURS).
3. The grammar-like form of the rules suggests using parsing techniques, extended to handle the highly ambiguous grammars that result from machine descriptions, and to choose least-cost parses.

4. If the compiler linearizes the tree into prefix form, it can use a string-matching algorithm to find the potential matches.

Practical tools implement each of the last three approaches. The compiler writer creates a description of a target machine's instruction set, and a generator creates executable code from the description.

The automated tools differ in details. The cost per emitted instruction varies. Some are faster, some are slower; none is slow enough to have a major impact on the speed of the resulting compiler. The approaches allow different cost models. Some systems restrict the compiler writer to a fixed cost for each rule; in return, they can perform some or all of the analysis during table generation. Others allow more general cost models where costs may vary during matching; these systems must manipulate those costs during instruction selection. In general, however, all these approaches produce instruction selectors that are both efficient and effective.

SECTION REVIEW

Instruction selection via tree-pattern matching uses trees to represent both operations in the code and operations on the target machine. The compiler writer creates a library of tree patterns that map constructs in the compiler's IR into operations on the target machine. Each pattern consists of a small IR pattern-tree, a code template, and a cost. In a single pass, the selector finds a locally optimal tiling for the tree. A second postorder walk generates the corresponding code from the templates associated with the tiles.

Several technologies have been used to implement tiling passes. These include hand-coded matchers, automata-based matchers, parser-based matchers operating on ambiguous grammars, and linear matchers based on algorithms for fast string matching. All of these technologies have worked well in one or more systems. The resulting instruction selectors run quickly and produce high-quality code.

REVIEW QUESTIONS

1. Tree-pattern matching seems natural for use in a compiler with a tree-like IR. How might sharing in the tree—that is, using a directed acyclic graph (DAG) rather than a tree—affect the algorithm?

Hint: Can the compiler recognize when pipelined loads can overlap?

2. The examples used a fixed cost for each pattern. How might the compiler use the ability to model costs dynamically?

11.5 ADVANCED TOPICS

Both tree-pattern matching and peephole instruction selectors have been designed for compile-time efficiency. Both techniques are limited, however, by the set of patterns that the compiler writer provides. To find the best such patterns, the compiler writer might consider using search techniques to discover the patterns.

The idea is simple. Combinations of instructions sometimes have surprising effects. Because the results are unexpected, they are rarely foreseen by a compiler writer and, therefore, are not included in the specification produced for a target machine. Automatic discovery can reveal new patterns.

Two distinct approaches that use exhaustive search to discover new patterns have appeared in the literature. The first has a peephole system discover new patterns as it compiles code. The second applies a brute-force search to the space of possible instructions.

11.5.1 Learning Peephole Patterns

To speed up peephole optimization, several systems have replaced the simplifier with a pattern-driven rewrite engine. The rewrite engine still operates over a small sliding window. It simply uses a library of patterns to recognize opportunities within the window rather than exploring all of the possible substitutions in each successive window. This approach decreases the time spent on simplification, but introduces a tradeoff between the size of the pattern set and the quality of the resulting code.

The tradeoff pits the compiler writer's time, spent developing the pattern set, against the quality of code that the instruction selector generates. A larger pattern set can perform more simplifications, but takes longer to write. A smaller pattern set can perform fewer simplifications, but takes less time to write.

Davidson and Fraser built a system that combined a fast pattern-driven simplifier with an exhaustive simplifier [126]. The compiler writer would provide an initial set of patterns to port the instruction selector to a new machine.

One effective way to generate the explicit pattern table needed by a fast, pattern-matching, peephole selector is to pair it with an optimizer that has a symbolic simplifier. Each time the simplifier discovers an improvement, it records both the initial sequence and the simplified sequence. A postpass checks these discovered patterns to ensure general applicability and records them for subsequent use.

By running the symbolic simplifier on a training set of applications, the compiler can discover most of the patterns it needs. Then, the compiler can use the table as the basis of a fast pattern-matching selector. This lets the compiler writer expend computer time during design to speed up routine use of the compiler. It greatly reduces the complexity of the patterns that must be specified.

Increasing the interaction between the two simplifiers can further improve code quality. At compile time, the fast pattern matcher will encounter some LLIR pairs that match no pattern in its table. When this occurs, it can invoke the symbolic simplifier to search for an improvement, bringing the power of search to bear only on the LLIR pairs for which it has no preexisting pattern.

To make this approach practical, the symbolic simplifier should record both successes and failures. This strategy allows it to reject previously seen LLIR pairs without the overhead of symbolic interpretation. When it succeeds in improving a pair, it should add the new pattern to the selector pattern table, so that future instances of that pair will be handled by the more efficient mechanism.

This learning approach to generating patterns has several advantages. It applies effort only on previously unseen LLIR pairs. It compensates for holes in the training set's coverage of the target machine. It provides the thoroughness of the more expensive system while preserving most of the speed of the pattern-directed system.

In using this approach, however, the compiler writer must determine when the symbolic optimizer should update the pattern tables and how to accommodate those updates. Allowing an arbitrary compilation to rewrite the pattern table for all users seems unwise; synchronization and security issues are sure to arise. Instead, the compiler writer might opt for periodic updates—storing the newly found patterns and releasing them as part of an update to the compiler.

11.5.2 Generating Instruction Sequences

The learning approach has an inherent bias: it assumes that the low-level patterns should guide the search for an equivalent instruction sequence. Some compilers have taken an exhaustive approach to the same basic problem. Instead of trying to synthesize the desired instruction sequence from a low-level model, they adopt a generate-and-test approach.

The idea is simple. The compiler, or compiler writer, identifies a short sequence of assembly-language instructions that should be improved. The compiler then generates all of the assembly-language sequences of cost one, substituting the original arguments into the generated sequence. It tests each one to determine if it has the same effect as the target sequence. When it has exhausted all sequences of a given cost, it increments the cost of the sequences and continues. This process continues until (1) it finds an equivalent sequence, (2) it reaches the cost of the original target sequence, or (3) it reaches an externally imposed limit on either cost or compile time.

While this approach is inherently expensive, the mechanism used for testing equivalence has a strong impact on the time required to test each candidate sequence. A formal approach, using a low-level model of machine effects, is clearly needed to screen out subtle mismatches, but a faster test can catch the gross mismatches that occur most often. If the compiler simply generates and executes the candidate sequence, it can compare the results against those obtained from the target sequence. This simple approach, applied to a few well-chosen inputs, should eliminate most of the inapplicable candidate sequences with a low-cost test.

This approach is, obviously, too expensive to use routinely or to use for large code fragments. In some circumstances, however, it merits consideration. If the application writer or the compiler can identify a small, performance-critical section of code, the gains from an outstanding code sequence might justify the cost of exhaustive search. For example, in some embedded applications, the performance-critical code consists of a single inner loop. Using exhaustive search for small code fragments—to improve speed or space—may be worthwhile.

Similarly, exhaustive search has been applied as part of the process of retargeting a compiler to a new architecture. This application uses exhaustive search to discover particularly efficient implementations for IR sequences that the compiler routinely generates. Since the cost is incurred when the compiler is ported, the compiler writer can justify the use of search by amortizing that cost over the many compilations that are expected to use the new compiler.

11.6 SUMMARY AND PERSPECTIVE

At its heart, instruction selection is a pattern-matching problem. The difficulty of instruction selection depends on the level of abstraction of the compiler's IR, the complexity of the target machine, and the quality of code desired from the compiler. In some cases, a simple treewalk approach will produce adequate results. For harder instances of the problem, however, the systematic search conducted by either tree-pattern matching or peephole optimization can yield better results. Creating a handcrafted treewalk instruction selector that achieves the same results would take much more work. While these two approaches differ in almost all their details, they share a common vision—the use of pattern matching to find a good code sequence among the myriad sequences possible for any given IR program.

Peephole selectors systematically simplify the IR program and match what remains against a set of patterns for the target machine. Because they lack explicit cost models, no argument can be made for their optimality. They generate code for a computation with the same effects as the IR program, rather than a literal implementation of the IR program. By contrast, tree-pattern matchers discover low-cost tilings by taking the low-cost choice at each decision point. The resulting code implements the computation specified by the IR program. Because of this subtle distinction in the two approaches, we cannot directly compare the claims for their quality. In practice, excellent results have been obtained with each approach.

The practical benefits of these techniques have been demonstrated in real compilers. Both GCC and LCC run on many distinct platforms. The former uses a peephole selector; the latter uses tree-pattern matching. The use of automated tools in both systems has made them easy to understand, easy to retarget, and, ultimately, widely accepted in the community.

Equally important, the reader should recognize that both families of automatic pattern matchers can be applied to other problems in compilation. Peephole optimization originated as a technique for improving the final code produced by a compiler. In a similar way, the compiler can apply tree-pattern matching to recognize and rewrite computations in an AST. BURS technology can provide a particularly efficient way to recognize and improve simple patterns, including the algebraic identities recognized by value numbering.

CHAPTER NOTES

Most early compilers used hand-coded, ad-hoc techniques to perform instruction selection [27]. With sufficiently small instruction sets, or large enough compiler teams, this worked. For example, the BLISS-11 compiler generated excellent code for the PDP/11, with its limited repertoire of operations [368]. The small instruction sets of early computers and minicomputers let researchers and compiler writers ignore some of the problems that arise on modern machines.

For example, Sethi and Ullman [321], and, later, Aho and Johnson [6], considered the problem of generating optimal code for expression trees. Aho, Johnson, and Ullman extended their ideas to expression DAGs [7]. Compilers based on this work used ad-hoc methods for the control structures. Ertl showed how to use tree parsing to find optimal code sequences for DAGs [153].

In the late 1970s, two distinct trends in architecture brought the problem of instruction selection to the forefront of compiler research. The move from 16- to 32-bit architectures precipitated an explosion in the number of operations and address modes that the compiler had to consider. For a compiler to explore even a large fraction of the possibilities, it needed a more formal and powerful approach. At the same time, the nascent Unix operating system began to appear on multiple platforms. This sparked a natural demand for C compilers and increased interest in retargetable compilers [217]. The ability to easily retarget the instruction selector plays a key role in determining the ease of porting a compiler to new architectures. These two trends started a flurry of research on instruction selection that started in the 1970s and continued well into the 1990s [78,79,143,170,176,297,298].

The success of automation in scanning and parsing made specification-driven instruction selection an attractive idea. Glanville and Graham mapped the pattern matching of instruction selection onto table-driven parsing [170,175,177]. Ganapathi and Fischer attacked the problem with attribute grammars [166].

The first peephole optimizer appears to be McKeeman's system [268]. Bagwell [31], Wulf et al. [368], and Lamb [246] describe early peephole systems. The cycle of expand, simplify, and match described in Section 11.3.1 comes from Davidson's work [125,128]. Kessler also worked on deriving peephole optimizers directly from low-level descriptions of target architectures [233]. Fraser and Wendt adapted peephole optimization to perform instruction selection [164,165]. The machine learning approach described in Section 11.5.1 was described by Davidson and Fraser [126].

Tree-pattern-matching code generators grew out of early work in table-driven code generation [10,46,177,195,249] and in tree-pattern matching [83,204]. Pelegrí-Llopart formalized many of these notions in the theory of BURS [290]. Subsequent authors built on this work to create a variety of implementations, variations, and table-generation algorithms [162,163,297]. The Twig system combined tree-pattern matching and dynamic programming [3,346].

Massalin proposed the exhaustive approach described in Section 11.5.2 [265]. It was applied in a limited way in the GCC compiler by Granlund and Kenner [180].

EXERCISES

Section 11.3

1. A peephole selector must deal with control-flow operations that include conditional branches, jumps, and labeled statements.

   a. What should a peephole selector do when it brings a conditional branch into the optimization window?

   b. Is the situation different when it encounters a jump?

   c. What happens with a labeled operation?

   d. What can the optimizer do to improve this situation?

2. Peephole selectors simplify the code as they select a concrete implementation for it. Assume that the peephole selector runs before instruction scheduling and register allocation and that the selector can use an unlimited set of virtual register names.

   a. Can the selector change the demand for registers?

   b. Can the selector change the set of opportunities for code reordering that are available to the scheduler?

Section 11.4

3. Build a low-level AST for each of the following expressions, using the tree in Fig. 11.1 as a model:

   a. y ← a × b + c × d

   b. w ← a × b × c - 7

   Assume that a is at offset 8 from the ARP; b and c are in registers (r_b and r_c); d is at offset 12 in the constant pool (which is at @CP); w is at offset 12 from the ARP; and y is held in a register (r_y).

   Use the rules from Fig. 11.5 to tile these trees.

4. In any treewalk instruction selection scheme, the compiler must choose an evaluation order for the subtrees. That is, at some binary node *n*, does it evaluate the left or the right subtree first?

   a. Does the choice of order affect the number of registers required to evaluate the entire subtree?

   b. How can this choice be incorporated into the bottom-up tree-pattern matching schemes?

5. The algorithm in Fig. 11.9 restricts rules to a single operator.

   a. Modify that algorithm so that it can correctly process two-operator rules such as $\mathtt{Reg} \rightarrow \blacklozenge\,(+\,(\mathtt{Reg}_1, \mathtt{NUM}_2))$.

   b. How do your changes affect the running time of the algorithm?

Chapter 12

Instruction Scheduling

ABSTRACT

The elapsed running time of a set of operations depends heavily on the order in which those operations are presented for execution. Instruction scheduling reorders the operations in a procedure to reduce the code's execution time. The set of legal orders for the operations is constrained by the need to preserve the flow of values from the original code. The goal is to reduce the number of cycles, start to finish, required to execute the code.

This chapter introduces the dominant technique for scheduling in compilers: greedy list scheduling in single basic blocks. Compiler writers use a variety of schemes to apply list scheduling to larger scopes. The chapter presents several of these, including superlocal scheduling, trace scheduling, and loop scheduling.

KEYWORDS

Instruction Scheduling, Dependence Graph, List Scheduling, Trace Scheduling, Software Pipelining

12.1 INTRODUCTION

On many processors, the order in which operations are presented for execution has a significant effect on the length of time it takes for a sequence of operations to execute. Different operations take a different number of machine cycles to complete. On a typical commodity microprocessor, integer addition and subtraction require fewer cycles than integer division; similarly, floating-point division takes more cycles than floating-point addition or subtraction. Multiplication usually falls between the corresponding addition and division operations. The cost of a load from memory depends on where in the memory hierarchy the loaded value resides at the time that the load issues.

To help the compiler manage this complex situation, most processors allow for overlapped execution of operations, albeit under constraints that ensure predictable answers. To allow operations to execute concurrently, most processors provide distinct and independent functional units. To decrease the time interval between the start of distinct operations, most processors use a

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00018-8

pipelined execution model. Both of these concurrency support mechanisms increase the complexity of the execution model.

Issue
A processor *issues* an operation when it begins the process of decoding and executing the operation.

A processor usually has a fixed number of *issue slots* per cycle, typically one per functional unit.

Instruction scheduling is the process of systematically reordering the operations in a block or a procedure so that they execute in fewer cycles. The scheduler takes in a partially ordered list of operations in the target machine's assembly language; it produces a fully ordered list of the same operations. It assumes that the code has already been optimized. It does not try to duplicate the optimizer's work. Instead, the scheduler packs operations into the available cycles and functional unit issue slots to reduce idle, or wasted, cycles.

Conceptual Roadmap

The order in which the processor issues operations has a direct impact on the speed of execution of that code. Thus, most compilers include an instruction scheduler to reorder the final operations in a way that improves performance. The scheduler's choices are constrained by the flow of data, by the delays associated with individual operations, and by the capabilities of the target processor. The scheduler must account for all these factors if it is to produce a correct and efficient schedule for the compiled code. Scheduling for single basic blocks is NP-complete under almost any realistic scenario.

The dominant technique for instruction scheduling is a greedy heuristic called list scheduling. List schedulers operate on straight-line code and use a variety of priority ranking schemes to guide their choices. Compiler writers have invented a number of frameworks to schedule over larger regions in the code than basic blocks; these regional and loop schedulers simply create conditions where the compiler can apply list scheduling to a longer sequence of operations.

A Few Words About Time

Instruction scheduling is a compile-time activity that tries to improve the quality of the code that the compiler produces. Like most optimizations, the scheduler runs at compile time. It analyzes the code and reorders the operations before they are emitted by the compiler. The benefits of that rearrangement accrue at runtime.

Overview

To speed up program execution, processor architects added features such as pipelined execution, multiple functional units, and out-of-order execution. These features make processors and their execution models more complex.

They make higher performance possible; they also make realized performance more sensitive to the order in which operations are issued. Thus, they make the instruction scheduler an important factor in the performance of the final compiled code.

Informally, instruction scheduling is the process whereby a compiler reorders the operations in the final code to decrease the number of cycles required for it to run. The instruction scheduler takes as input a partially ordered list of instructions; it produces as output an ordered list of instructions constructed from the same set of operations. The scheduler assumes a fixed set of operations; it does not optimize the code. The scheduler assumes that values in registers remain in registers and values in memory remain in memory; it does not change storage allocation or storage placement decisions.

The scheduler's primary goal is to compute an issue order for the operations in a block such that (1) operands are available when needed, (2) the functional units are well utilized, and (3) the number of cycles required to execute the block is minimized.

The scheduler tries to minimize the block's running time. It cannot, in practice, guarantee an optimal schedule.

Example

Consider the example in Fig. 12.1(a) that computes `(a + a) × b × c × d`; this example first appeared in Section 1.3.3. We have substituted concrete numbers for the offsets `@a`, `@b`, `@c`, and `@d`. The column labeled "Start" shows the cycle in which each operation issues. Assume that the processor has one functional unit; loads and stores take three cycles; a multiply takes two cycles; and all other operations complete in a single cycle. The original code, shown in panel (a), executes in 22 cycles, known as it *schedule length.*

Schedule length
For a code fragment, its *schedule length* is the number of cycles needed for it to complete execution.

The scheduled code, shown in panel (b), executes in 13 cycles. It separates long-latency operations from operations that reference their results. This separation allows operations that do not depend on the results of those long-latency operations to execute in parallel before those operations finish. The scheduled code issues load operations in the cycles 1, 2, and 3; the results are available in cycles 4, 5, and 6, respectively. The overlap among operations effectively hides the latency of the memory operations. The same approach hides the latency of the first multiply. The reordered code has a schedule length of 13 cycles, a 41 percent improvement while executing the same operations. The increased concurrency requires one extra register.

Roadmap

This chapter examines both the instruction scheduling problem and the algorithms that compilers use to solve it. Section 12.2 discusses both the processor features that make scheduling hard and the ideas that underlie scheduling

Start	Operations			
1	loadAI	r_{arp}, 4	$\Rightarrow$	r_1
4	add	r_1, r_1	$\Rightarrow$	r_1
5	loadAI	r_{arp}, 8	$\Rightarrow$	r_2
8	mult	r_1, r_2	$\Rightarrow$	r_1
10	loadAI	r_{arp}, 12	$\Rightarrow$	r_2
13	mult	r_1, r_2	$\Rightarrow$	r_1
15	loadAI	r_{arp}, 16	$\Rightarrow$	r_2
18	mult	r_1, r_2	$\Rightarrow$	r_1
20	storeAI	r_1	$\Rightarrow$	r_{arp}, 4

(a) Original Code

Start	Operations			
1	loadAI	r_{arp}, 4	$\Rightarrow$	r_1
2	loadAI	r_{arp}, 8	$\Rightarrow$	r_2
3	loadAI	r_{arp}, 12	$\Rightarrow$	r_3
4	add	r_1, r_1	$\Rightarrow$	r_1
5	mult	r_1, r_2	$\Rightarrow$	r_1
6	loadAI	r_{arp}, 16	$\Rightarrow$	r_2
7	mult	r_1, r_3	$\Rightarrow$	r_1
9	mult	r_1, r_2	$\Rightarrow$	r_1
11	storeAI	r_1	$\Rightarrow$	r_{arp}, 4

(b) Scheduled Code

■ **FIGURE 12.1** Example Block from Chapter 1.

algorithms. Section 12.3 introduces list scheduling, the standard framework that most compilers use. Section 12.4 examines techniques that compiler writers use to apply that framework to larger domains. The Advanced Topics section examines software pipelining, a technique for scheduling loops.

12.2 BACKGROUND

Schedule length (*again*)
A block's schedule length is the number of cycles that elapse from when the first operation issues until the last operation completes.

To compensate for the complex execution models of modern processors and to capitalize on them, compilers systematically reorder the operations in the code that they generate. The compiler has multiple goals in this process. It tries to decrease the number of empty or wasted operation issue slots. It tries to decrease idle time on the individual functional units. Finally, it tries to reduce the overall schedule length.

Dependence graph
The dependence graph $\mathcal{D}$ for block *b* has a node for each operation in *b* and an edge (*x*,*y*) if operation *y* uses the result of *x*—that is, *y* depends on *x*.

An instruction scheduler analyzes the code to determine how the flow of values constrains the order of execution. It builds a *dependence graph*, $\mathcal{D}$, to represent the order constraints induced by the flow of values. The scheduler reorders the operations, within the constraints expressed in $\mathcal{D}$, to capitalize on processor and system resources while hiding, to the extent possible, processor and system latencies.

To design and implement a scheduler, the compiler writer must understand the processor's feature set, the impact that those features have on the performance of compiled code, and the ways in which those features present either challenges or opportunities to the scheduler. This section describes some of the processor features that complicate scheduling. It then introduces the scheduling problem.

12.2.1 Architectural Features That Affect Performance

Computer architects have explored and implemented myriad features intended to increase peak processor performance. Some features, such as pipelined execution, both affect scheduling and rely on scheduling. Others, such as register windows, have a negligible impact on the design and effectiveness of a scheduler. This section describes three processor features that interact with scheduling: pipelined execution, variable latency operations, and multiple functional units.

This section provides an overview of features that affect scheduling. For more detail the reader should consult a textbook on computer architecture [200].

Unless otherwise stated, we assume that all operations are *nonblocking* operations—the processor can issue operations to a functional unit in each cycle, even if earlier operations are not yet complete. Operations begin execution when issued, unless the processor detects that the operation uses a value that will be defined by another operation that is still executing. When the processor detects such a *pipeline data hazard*, it stalls the operation until its operand is available.

Pipelined Execution

The fundamental unit of time in scheduling is one cycle of the processor's clock. Many operations, such as an integer add, an immediate load, or a logical shift, operate in a single cycle. More complex operations, such as multiply, divide, branch, jump, and most memory operations, need more than one cycle to complete. A typical processor can issue one operation per cycle to each functional unit, so a short cycle time leads to more operations issued per second.

Multicycle operations are often *pipelined*; that is, the operation is decomposed into a series of simpler, single-cycle steps that, in sequence, perform the complex operation. The pipeline resembles an assembly-line: each pipeline stage performs part of the operation; at the end of the cycle, the result passes to the next pipeline stage.

A typical operation reads its operands at the start of the cycle in which it issues and writes its results at the end of the cycle in which it completes.

To exploit pipelined execution, the scheduler must understand when in the pipeline an operation reads the values that it uses and when it writes the values that it defines. Ideally, the scheduled code issues each operation after its operands are available. To allow the scheduler to fill all of the issue slots while delaying each operation until its arguments are ready, the code must have sufficient instruction-level parallelism (ILP).

The optimizer can increase the amount of exposed instruction level parallelism in the code. See, for example, Section 8.4.2.

Variable Latency Operations

Some operations have unpredictable latencies. The latency of a load operation or a store operation, for example, depends on the state of the processor's memory hierarchy. Some implementations of multiplication and division

WHAT HAPPENS WHEN AN OPERAND IS NOT READY?

In some situations, a processor will try to execute an operation before all of its operands are ready. The following sequence demonstrates the problem. Assume that the processor issues the `mult` and the `add` in consecutive cycles.

```
mult  r1,r2 ⇒ r3
add   r1,r3 ⇒ r4
```

If `add` takes one cycle and `mult` takes three cycles, then the value of r_3 is not available when the `add` issues. The processor can detect such a *data hazard* before it issues an operation by comparing the names of the arguments to the names that will be defined by operations that are still in execution.

Most processors stall the new operation until the argument is available. Processors differ, however, on the impact of a stall.

- *Statically Scheduled, or In-Order, Processors* Once the need for a stall is detected, these processors stop issuing additional operations until the hazard clears. Some processors stall just the one functional unit; others stall all the functional units.
- *Dynamically Scheduled, or Out-of-Order, Processors* These processors maintain a window into the instruction stream and issue operations as their arguments become ready, with preference to operations early in the window.

The scheduler should place operations into cycles where they can execute without a stall. In the example, it could schedule two independent operations between the `mult` and the `add`, if such operations were available.

A few processors have relied on the compiler to detect and mitigate data hazards. While this approach may simplify the hardware, it can complicate the scheduler. In such a scheme, the compiler may need to insert `nop`s to enforce temporal separation between operations.

have small variations in latency based on the bit-patterns of their arguments. If the operation has a maximum latency, the scheduler can use that latency and produce a safe schedule.

For memory operations, the maximum latency is often impractically long for use in scheduling. Consider a tight eight-operation loop that contains a load operation. If the load can take anywhere from two cycles (a hit in the first-level cache) to 100 cycles (a miss to RAM), then using the 100 cycle latency will produce many idle cycles if the load operations exhibit even modest locality. Thus, the scheduler should estimate some reasonable expected latency and rely on the hardware to detect incomplete operations and stall their successors.

Multiple Functional Units

To increase the number of operations per second that the processor can execute, architects build processors with multiple functional units. Each functional unit can initiate execution of an operation in each cycle. To keep multiple functional units busy, the compiler must issue multiple independent operations in the same cycle, again exploiting ILP. The scheduler must understand both the resources available—the number of functional units and the sets of operations that each can execute—and the dispatch discipline that determines how the processor issues operations.

Superscalar processors have a single instruction stream. At each cycle, the processor looks at the instruction stream and issues as many consecutive operations as it can. Consider a simple two functional unit, in-order processor. At each cycle, it would pick the first operation in the window and issue it. It would then examine the next operation; if that operation can issue on the unused functional unit, the processor issues it. If not, the functional unit remains idle in the current cycle. An out-of-order processor has a larger window and looks at successive operations until it finds one that can issue without a hazard.

Superscalar
a processor that has one instruction stream and multiple functional units, and that issues multiple operations per cycle

By contrast, a *very-long instruction-word* (VLIW) processor manages the operation issue process by encoding one operation per functional unit into each instruction. This strategy simplifies the process of selecting and issuing operations. Instead, the compiler must fully specify the operation-issue behavior of the program. A VLIW instruction may represent idle operations (nops) implicitly or explicitly.

VLIW processor
A *very-long instruction-word* processor has an instruction format with a slot for each functional unit in each cycle.

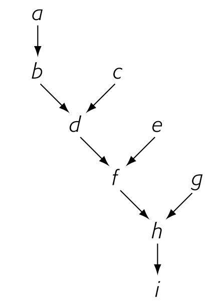
A *packed* VLIW processor has a variable-length instruction word that can hold up to one operation for each functional unit.

Many variations exist between these two endpoints. Superscalar processors vary from strictly in-order processors to out-of-order processors with large lookahead windows. Some VLIW processors have long instruction words, but allow the instructions to vary in length to save code space. In order to plan execution time behavior, the scheduler must understand the processor's instruction dispatch discipline.

The diversity of dispatch mechanisms blurs the distinction between an operation and an instruction. On VLIW and packed VLIW machines, an instruction contains multiple operations. On superscalar machines, we usually refer to a single operation as an instruction and describe these machines as issuing multiple instructions per cycle. Throughout this book, we use the term *operation* to describe a single opcode and its operands. We use the term *instruction* to refer to an aggregation of one or more operations that all issue in the same cycle. In deference to tradition, we still refer to this chapter's problem as *instruction scheduling*, although it might be more precisely called *operation scheduling*.

```
a:  loadAI   rarp, 4   ⇒ r1
b:  add      r1, r1    ⇒ r1
c:  loadAI   rarp, 8   ⇒ r2
d:  mult     r1, r2    ⇒ r1
e:  loadAI   rarp, 12  ⇒ r2
f:  mult     r1, r2    ⇒ r1
g:  loadAI   rarp, 16  ⇒ r2
h:  mult     r1, r2    ⇒ r1
i:  storeAI  r1        ⇒ rarp, 4
```

(a) Example Code



(b) Its Dependence Graph

■ **FIGURE 12.2** Dependence Graph for the Example.

12.2.2 The Instruction Scheduling Problem

The instruction scheduling problem is defined over a basic block and its dependence graph, $\mathcal{D}$. Nodes in $\mathcal{D}$ represent operations in the block. Edges in $\mathcal{D}$ represent the flow of values between those operations. Edges are directed; the edge (x,y) indicates that operation y uses the value produced by operation x. Additionally, each node has two attributes, a *type* and a *delay*. For a node n, the operation corresponding to n must execute on a functional unit specified by $type(n)$. That operation requires $delay(n)$ cycles to complete. Fig. 12.2 shows the code for our running example and its dependence graph.

Edges in $\mathcal{D}$ show the flow of values. If we reversed the edges, $\mathcal{D}$ would form a forest of DAGs.

We draw $\mathcal{D}$ with roots at the bottom, so that the drawing reflects the order of the code.

$\mathcal{D}$ is a forest of reversed directed acyclic graphs (DAGs). Thus, we use DAG terminology to describe $\mathcal{D}$. Nodes without predecessors, such as a, c, e, and g in the example, are *leaves*. Leaves can be scheduled as early as possible, because they depend on no other operations. Nodes without successors, such as node i in the example, are *roots*. Roots can only be scheduled after all of their predecessors have completed.

Assume, for the moment, that all operations have known delays. A discussion of scheduling with variable delays starts on page 636.

Given a dependence graph $\mathcal{D}$ for a block, a schedule S maps each node n to a nonnegative integer that denotes the cycle in which it should be issued, assuming that the first operation issues in cycle 1. Thus, the ith instruction contains the set of operations $\{n \mid S(n) = i\}$. A valid schedule must meet three constraints.

1. $S(n) \geq 1$, for each $n \in N$. This constraint forbids operations that issue before execution starts. A schedule that violates this constraint is not well formed. For the sake of uniformity, the schedule must also have at least one operation n' with $S(n') = 1$.

2. If $(n_1, n_2) \in E$ then $S(n_1) + delay(n_1) \le S(n_2)$. This rule enforces correctness. An operation should not issue until its operands have been defined. Some processors expect the compiler to insert nops to enforce this constraint. Others relax it and use hardware interlocks to delay an operation until its operands are available.
3. Each instruction contains no more operations of each type t than the target machine can issue in a cycle. This constraint enforces feasibility. It ensures that no instruction over-subscribes the functional units.

Given a well-formed, correct, and feasible schedule, the schedule length is simply the cycle number in which the last operation completes, assuming the first instruction issues in cycle 1. Schedule length can be computed as:

$$L(S) = \max_{n \in N} \; (S(n) + delay(n) - 1).$$

If *delay* accurately captures the operation latencies, schedule S should execute in $L(S)$ time. (Variable latency operations might introduce inaccuracies into *delay*.) A schedule S_i is time optimal if $L(S_i) \le L(S_j)$ for all other schedules S_j that contain the same set of operations.

Schedule Quality

Schedule length is the classic measure of schedule quality. With fixed latency operations, schedule length accurately reflects execution time. With variable latency operations, the actual cost of the operation depends on dynamic factors. As long as the scheduler uses consistent assumptions, the lengths of different schedules should be comparable.

Schedules can be measured in terms other than time. Two schedules S_i and S_j for the same block might produce different demands for registers If the processor requires the scheduler to insert nops for idle functional units, then two schedules may differ in the number of operations fetched. Finally, S_j might require less energy than S_i to execute on the target system because it never uses one of the functional units, it fetches fewer instructions, or it causes fewer bit transitions in the processor's fetch and decode logic.

What Makes Scheduling Hard?

The fundamental task of a scheduler is to assign each operation in the block a cycle in which it will issue. For each cycle, it builds a set of operations. As it does so, the scheduler must ensure that each operation issues only when its operands are available. If the processor has hardware interlocks to prevent an operation from reading an undefined value, then the schedule can issue an operation earlier—once all the operations that compute its operands have issued. That operation, however, will stall until the operands are ready.

MEASURING RUNTIME PERFORMANCE

The primary goal of instruction scheduling is to improve the running time of the generated code. Because this chapter covers scheduling algorithms, our focus is on schedule length. However, discussions of performance use many different metrics.

- *Instructions per Second* This metric measures operations retired per second. Its common use is to state hardware peak performance; it can be used to state application performance.
- *Cycles per Instruction* (CPI) This metric divides elapsed time, in cycles, by operations retired. For a fixed stream of operations, a smaller CPI indicates better performance.
- *Benchmarks* These metrics measure the elapsed time to complete a known set of tasks. They provide information about overall system performance on a particular workload.

Performance measures provide insight into the quality of code generated by the compiler. They do not, in general, provide enough detail to allow compiler writers to determine where to focus their effort. A high CPI value might arise from the structure of the code, from a poor scheduling heuristic, or from insertion of excessive spill code by the register allocator. (Spill operations often have multicycle latencies.)

To manage processor resources, the scheduler faces conflicting goals. To achieve high performance it should keep the functional units busy. To hide the latencies of multicycle operations, it should overlap the execution of independent operations. The scheduler should schedule variable-latency operations, such as loads, early relative to their uses. Each of these goals is important; all consume the same finite resource: available ILP in the code. The scheduler must tradeoff between these competing goals.

When the scheduler places an operation i in cycle c, that decision affects the earliest possible placement of any operation that relies on the result of i—any operation in $\mathcal{D}$ that is reachable from i. If more than one operation can legally execute in cycle c, then the scheduler's choice can change the earliest placement of many operations—all those operations dependent (either directly or transitively) on each of the possible choices.

Local instruction scheduling is NP-complete for all but the simplest architectures. In practice, compilers approximate good solutions to scheduling problems using greedy heuristics. Most of the scheduling algorithms used in compilers are based on a single family of heuristic techniques, called *list scheduling*. The following section describes list scheduling in detail. Subsequent sections show how to extend the paradigm to larger scopes.

SECTION REVIEW

The advent of pipelined execution, nonblocking operations, variable latency operations, and multiple functional units makes code performance sensitive to the order in which operations are issued. The compiler's instruction scheduler analyzes the dependence and latency structure of the code and reorders operations to improve execution time while preserving the code's original meaning.

Instruction scheduling is a hard problem that involves low-level details of the target architecture. It is also a computationally complex problem. Instruction scheduling straight-line code is NP-complete for almost any realistic architecture. Thus, most compilers use a greedy heuristic technique, list scheduling, to solve the problem.

REVIEW QUESTIONS

1. The Texas Instruments C6X series of processors introduced a multicycle `nop`. The operation took a single argument: the number of consecutive cycles that it should idle the functional unit.

   What advantages might a multicycle `nop` provide? How might the scheduler capitalize on the availability of this operation?

2. Assume that load operations are nonblocking with latency k, so that one load takes k cycles, but two loads, issued in successive cycles, take $k+1$ cycles.

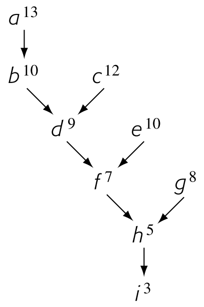
   Should this observation affect *instruction selection*? How might the algorithms in Chapter 11 account for this effect?

12.3 LOCAL SCHEDULING

List scheduling is a greedy, heuristic paradigm. Since the late 1970s, it has been the dominant technique that compilers use to schedule instructions, largely because it discovers reasonable schedules and it adapts easily to new ISAs. However, list scheduling is an approach rather than a specific algorithm. Wide variation exists in both implementations and detailed heuristics. This section explores the basic list scheduling framework, as well as a couple of variations on the scheme.

List scheduling operates on a block's dependence graph. It annotates $\mathcal{D}$ with latency and priority information. It then uses the annotated graph to construct, cycle by cycle, a schedule for the block that the graph represents.

Critical path
the longest latency path through $\mathcal{D}$



Dependence Graph
Annotated with Latencies

The dependence graph captures most of the schedule-critical properties of a block. The figure in the margin repeats the graph from Fig. 12.2(b). For a node n, the path length from n to the root is shown as a superscript on n. (If $\mathcal{D}$ had multiple roots, the scheduler would use the maximal path length to a root.) The superscripts show that the path *abdfhi* is the *critical path* through $\mathcal{D}$; it determines the block's minimal execution time.

How, then, should the compiler schedule this computation? Each operation should be placed in a cycle where its operands are available. Since a, c, e, and g are leaves, they are the initial candidates for scheduling. The fact that a lies on the critical path suggests that it be scheduled into the first cycle. Once a has been scheduled, the longest path remaining in $\mathcal{D}$ is *cdfhi*, suggesting that c issue in the second cycle. Given the prefix *ac*, b and e tie for latency. However, b depends on the result of a, which will not be available until cycle four. Thus, *eb* is a better suffix to *ac* than is *be*. Continuing in this fashion leads to the schedule *acebdgfhi*, the schedule shown in Fig. 12.1(b).

To fix this conflict, the code in Fig. 12.1(b) renamed the value defined by e and used by f from r_2 to r_3.

Simply choosing that schedule, however, is not enough The operations for nodes c and e both define r_2; d, in turn, uses the value from c. The scheduler cannot move e before d unless it renames the result of e to avoid the conflict with c's definition of r_2. The dependence graph fails to capture this conflict because it arises from the fact that executing e overwrites the value, rather than from the flow of a value.

Antidependence
Operation x is *antidependent* on operation y if x precedes y and y defines a value used in x. Reversing their order of execution could cause x to compute a different value.

Ordering constraints such as the one between c and e are called *antidependences*. We denote the antidependence between e and d as $e \rightarrow_A d$. The scheduler must respect the antidependences in the original code. The example contains four antidependences, namely, $e \rightarrow_A c$, $e \rightarrow_A d$, $g \rightarrow_A e$, and $g \rightarrow_A f$. All of them involve redefinition of r_2. (Constraints exist based on r_1 as well, but each antidependence on r_1 duplicates a dependence based on the flow of values.)

SSA form creates a name space that avoids kills and, thus, many antidependences.

The scheduler has two options to handle this kind of antidependence. It can either represent antidependences explicitly and respect them in the same way that it handles dependences from the flow of values, or it can rename around them. The former approach restricts the set of schedules that the compiler can discover. The latter approach increases the size of the name space, which may force the register allocator to insert spill code into the block. The local scheduling algorithm that we present assumes that the scheduler will rename register-based values in the code. The section on dependence graph construction will provide more details on detecting and representing antidependences.

12.3.1 The Algorithm

Classic list scheduling operates on a single basic block. To schedule a single block, the scheduler follows a four-step plan.

1. *Rename* To avoid antidependences on register-based values, the compiler renames them. This step is not strictly necessary, but it simplifies the scheduler's implementation.
2. *Build a Dependence Graph* The scheduler walks the block from bottom to top. At each operation, it builds a node that represents both the operation and the value it defines. It then connects that node to the node for each operation that uses the value.
3. *Assign Priorities to Each Operation* To guide the choice of operations, the scheduler computes one or more priorities for each node in $\mathcal{D}$. Priorities are typically computed in a walk over $\mathcal{D}$. The maximum latency-weighted distance to a root is a common priority scheme.
4. *List Scheduling* The scheduler starts in the block's first cycle and chooses as many operations as possible to issue in that cycle. It then increments its cycle counter, updates its notion of which operations are ready to execute, and schedules the next cycle. It repeats this process until each operation has been scheduled.

The following subsections explore each of these steps in more detail.

12.3.2 Renaming

The first step in the algorithm renames all unambiguous scalar values in the block so that each name corresponds to a single definition. As a matter of code shape, we assume that values in registers are unambiguous from their definition to their last use. The renaming algorithm operates on these register names. To denote the different name spaces, a name in the original code is a *source* name while a name created by renaming is a *virtual* name.

Recall that a value is ambiguous if the code can reach it through multiple names (see Section 4.7.1).

The compiler can create a virtual-name space with properties that simplify the task at hand. For example, local value numbering creates a virtual name space where name identity implies value identity. For scheduling, the renaming algorithm creates a name space that eliminates antidependences between unambiguous scalar values. That name space, in turn, simplifies the later phases of scheduling.

In a single block, each definition creates a new value. Renaming finds each such value and assigns it a new virtual name. Within the span between the value's definition and its last use, the algorithm rewrites references to the source name with its virtual name. This process creates a one-to-one correspondence between values and names.

```
VName ← 0
for i ← 0 to max source-register number do
    SToV[i] ← invalid                    // initialization
for each Op in the block, bottom to top do
    for each definition, O, in Op do     // do defs first
        if SToV[O] = invalid then        // invalid def indicates
            SToV[O] ← VName++            // an unused value
        O ← SToV[O]                      // O gets its new name
        SToV[O] ← invalid                // next ref is a new name
    for each use, O, in OP do            // do uses second
        if SToV[O] = invalid then        // start a new value
            SToV[O] ← VName++
        O ← SToV[O]                      // O gets its new name
```

■ **FIGURE 12.3** Renaming for List Scheduling.

This algorithm also appears in local register allocation (see Section 13.3).

The algorithm, shown in Fig. 12.3, discovers values, assigns virtual names, and rewrites references in a single backward pass over the block. Its central data structure is a map, *SToV*, from source names to virtual names. The algorithm begins by setting all the *SToV* entries to invalid and initializing the first virtual name to zero. It then iterates over each operation in the block, from bottom to top.

In the example, note that all of the names in a `store` are uses, not definitions.

At each operation, the algorithm processes definitions before uses. It rewrites defined names with their virtual names; if a defined name has no virtual name, its value is not used in the block. It invalidates a virtual name when it encounters the name's definition, which forces the next use of that source name to have its own virtual name. For each of the used names, it checks *SToV* to determine if the value has a virtual name; if not, it assigns the value the next virtual name and rewrites it.

The algorithm processes definitions before uses to ensure correct behavior when a single operation uses a name and then redefines it. For example, in an operation such as `addI` r_{18}, 12 ⇒ r_{18}, the algorithm will first rewrite the definition with *SToV*[r_{18}]. Next, it will invalidate *SToV*[r_{18}]. Then, it will create a new virtual name and update *SToV*[r_{18}] with the new name. Finally, it will rewrite the use with the new name, from *SToV*[r_{18}].

```
loadAI   r_arp, 4   ⇒ r7
add      r7, r7     ⇒ r5
loadAI   r_arp, 8   ⇒ r6
mult     r5, r6     ⇒ r3
loadAI   r_arp, 12  ⇒ r4
mult     r3, r4     ⇒ r1
loadAI   r_arp, 16  ⇒ r2
mult     r1, r2     ⇒ r0
storeAI  r0     ⇒ r_arp, 4
```

Example After Renaming

After renaming, each live range has a unique virtual name. A source name that was defined in multiple operations will be rewritten with multiple distinct virtual names. The renamed version of the example block appears in the margin. The antidependences that arose from order of operations in the input program have been broken; the actual flow of values remains the same.

As discussed in the next subsection, r_{arp} has not been renamed because it is live across the block boundaries.

The renaming algorithm also plays an important role in local register allocation (see Section 13.3).

End-of-Block Conditions

The algorithm in Fig. 12.3 assumes that the block has no surrounding context—that is, the block is the entire program. In most situations, a block b has both predecessors and successors in the control-flow graph. For the generated code to function correctly in that context, the renaming algorithm must preserve source names that are live across those boundaries.

Before renaming, the compiler can compute LIVE information (see Sections 8.6.1 and 9.2.2). The algorithm then needs three modifications to deal with the boundary conditions. It should initialize *VName* to a value higher than the maximum source-register number. To handle successor blocks, it should initialize *SToV* to i for any name $i \in$ LIVEOUT(b). These two steps ensure that values in LIVEOUT have the names expected by successor blocks.

Names are assumed to be represented as integers. The compiler can use as many virtual names as it needs.

Handling boundaries with predecessor blocks is slightly more complex. Before the algorithm assigns a new *VName* to some source name, it must determine whether that value is defined in the current block or in some predecessor block. Because the input code can define names multiple times, checking LIVEIN(b) is insufficient. Instead, the compiler must determine whether the current use is upward exposed.

During LIVE analysis, the compiler computes the set of upward exposed variables in b, UEVAR(b). As it does so, it can mark any upward exposed uses. Then, when the renaming algorithm encounters a use of name i such that *SToV*[i] is invalid, it can assign i to *SToV*[i] if the use is upward exposed, rather than *VName*.

12.3.3 Building the Dependence Graph

The second step in the local scheduling process builds a dependence graph, $\mathcal{D}$, for the basic block. The algorithm, shown in Fig. 12.4 is straightforward. It creates an empty map, M, that takes virtual names from the code into nodes in $\mathcal{D}$. It then iterates over the operations in the block, from top to bottom. For each operation, O, it creates a node n in $\mathcal{D}$. It updates M so that $M(d) = n$, for each name d defined by O. It then adds edges from n to the nodes that define the values used in O.

If names are small integers, M can be implemented as a vector.

The final step adds edges to represent antidependences through memory locations and ensure that the final schedule respects them. The idea is simple.

Recall that renaming eliminated antidependences among unambiguous scalar names.

```
create an empty map, M                      // definitions to nodes
create a node, undef, in $\mathcal{D}$                  // for an undefined value

for each operation O, top to bottom do      // walk the block
    create a node n for O, in $\mathcal{D}$             // n represents O
    for each name, d, defined in O do
        set M(d) to n
    for each name, u, used in O do          // true dependence
        if M(u) is undefined then
            set M(u) to undef
        add an edge (n,M(u)) to $\mathcal{D}$
    if O is a memory operation then         // antidependences
        add serialization edges as needed
```

■ **FIGURE 12.4** Building the Dependence Graph After Renaming.

If an operation might change the contents of memory, the scheduler must ensure that the operation does not affect values seen by other operations. The details are slightly complex.

```
load   r17 ⇒ r5
load   r18 ⇒ r6
store  r7  ⇒ r17
```

Consider, for example, the three operation code fragment shown in the margin. The scheduler can reorder the two loads; neither modifies the contents of the memory so changing their relative issue order has no effect on the values placed in r_5 and r_6. The store, however, changes the contents of memory. In the absence of information about the addresses in the loads and stores, the scheduler must ensure that the loads read from memory before the store writes to memory. The compiler may need to add edges to $\mathcal{D}$ to *serialize* these operations.

Serialization edge
An edge that represents an antidependence is a *serialization* edge.

The first load uses the same address as the store; the compiler must schedule these operations so that the load reads its value before the store overwrites that value. The second load uses a different register as its address. If the compiler can prove that r_{17} and r_{18} always have distinct values, then the scheduler can reorder the store and the second load. If the compiler cannot prove those values distinct, it must preserve the relative ordering of the second load and the store, as well.

The table in Fig. 12.5 summarizes the situations that can arise. The scheduler can always swap the order of two loads. With a load and a store, it must preserve their original order; it must *serialize* the operations. The table denotes this delay as serial. With a store followed by a load, the scheduler must ensure that the store completes before the load issues; in general, it must allow the full latency of the store.

Conflict Type		First Op	Second Op	Delay
Read After Read	(RAR)	`load`	`load`	none
Write After Read	(WAR)	`load`	`store`	serial
Read after Write	(RAW)	`store`	`load`	store
Write after Write	(WAW)	`store`	`store`	serial

■ **FIGURE 12.5** Serialization Latencies for Memory Operations.

Given a pair of memory references, the compiler must enforce the appropriate delay, unless it can prove that the two operations refer to disjoint locations in memory. In terms of dependence graph construction, the compiler must add an edge to represent these constraints, weighted with the appropriate latency. The compiler can reduce the number of antidependences that it must represent by analyzing the values of the memory addresses.

12.3.4 Computing Priorities

To guide scheduling decisions, the compiler computes one or more priority values for each operation. When the scheduler confronts a choice, it takes the operation with the highest priority value. The scheduler needs a tie-breaking strategy; a common approach uses a second or third priority value. The literature discusses many priority schemes; one often used scheme is the operation's maximum latency-weighted distance to a root in $\mathcal{D}$.

Priorities such as latency-weighted depth, number of descendants, breadth-first order, and depth-first order can be computed in a traversal of $\mathcal{D}$. The scheduler can incorporate other heuristics, such as a preference for loads over stores, with simple numerical manipulations of the priority scores.

Multiple priorities can be accommodated as a linear combination. If p_1 and p_2 are scores from different priority metrics, the scheduler can combine them as $\alpha \cdot p_1 + \beta \cdot p_2$ to create a single value. If $\alpha \gg \beta$, the effect is to use p_2 to break ties in the p_1 metric. Alternatively, the compiler can restrict $\alpha + \beta = 1$ and tune the values of α and β for the best result across a training set of codes.

Using lower-weight priorities to break ties can push the schedule toward specific goals. For example, using the operation's latency as a lower-weight priority pushes long latency operations earlier in the block. Using the number of operands that are last uses as a lower-weight priority can decrease demand for registers.

Some priority schemes are hard to encode as numerical weights. For example, if the processor restricts certain operations to specific functional units,

```
Cycle $\leftarrow$ 1
Ready $\leftarrow$ leaves of $\mathcal{D}$
Active $\leftarrow$ $\emptyset$

while (Ready $\cup$ Active $\neq \emptyset$) do
    for each functional unit, f, do
        if there is an op in Ready for f then
            let O be the highest priority op      // choose O by priority
              in Ready that can execute on f

            remove O from Ready                   // schedule O in Cycle
            S(O) $\leftarrow$ Cycle
            Active $\leftarrow$ Active $\cup$ {O}

    Cycle $\leftarrow$ Cycle + 1                        // start next Cycle

    for each O $\in$ Active do
        if S(O) + delay(O) $\leq$ Cycle then          // update Ready list
            remove O from Active
            for each successor s of O in $\mathcal{D}$ do
                if s is ready
                    then add s to Ready
```

■ **FIGURE 12.6** The List-Scheduling Algorithm.

the compiler writer may want to boost the priority of those operations on their respective units. Rather than manipulate the priority numbers, the compiler writer can track the restricted and unrestricted operations separately. The scheduler can first draw from the restricted list and then, if needed, from the unrestricted list.

12.3.5 **List Scheduling**

The final step in scheduling is to construct the actual schedule for the block. The most widely used algorithm for this part of the process is the list-scheduling algorithm—a greedy heuristic that, in practice, constructs good schedules. Fig. 12.6 shows the basic algorithm.

The algorithm performs an abstract simulation of the block's execution. It ignores the details of values and operations to focus on the timing constraints imposed by edges in $\mathcal{D}$. To track time, it maintains a simulation clock, in the variable *Cycle*. It initializes *Cycle* to 1 and increments *Cycle* as it proceeds through the block.

The algorithm uses two lists to track operations. The *Ready* list holds all the operations that are "ready" to execute in the current cycle; that is, their operands have already been computed. Initially, *Ready* contains all the leaves of $\mathcal{D}$, since they do not depend on other operations in the block. The *Active* list holds all operations that were issued in an earlier cycle but have not yet completed execution.

The list-scheduling algorithm follows a simple discipline. Each iteration of the scheduler begins in some specific cycle. It examines the *Ready* list and tries to select an operation for each functional unit. If more than one operation is available for some unit, it chooses the operation with the highest priority score. After it has considered all units, the scheduler increments *Cycle* and updates the *Ready* list.

To update *Ready* for the new cycle, the scheduler first finds all of the operations in *Active* that completed in the previous cycle. For each such completed operation, the scheduler checks each successor s in $\mathcal{D}$ to determine if s is now ready to execute. More precisely, it determines if all of the operands used by s are available—that is, the operation that produced the operand has, itself, completed execution.

Operation O completes at the end of cycle $S(O)$ - *first* + delay(O).

The process terminates when the simulated clock reaches the first cycle where every operation has completed. If all operands of the leaves of $\mathcal{D}$ are available in the first cycle, and *delay(op)* accurately reflects the execution time of *op*, then the simulated clock should match the actual execution time, in cycles. A simple postpass can emit the operations in order and insert `nops` if needed.

The algorithm must respect one final constraint. Any block-ending branch or jump must be scheduled so it does not complete execution before the block ends. If i is the block-ending branch, it cannot be scheduled earlier than cycle $L(S) + 1 - delay(i)$. Thus, a single-cycle branch must be scheduled in the last cycle of the block, and a two-cycle branch must appear no earlier than the second to last cycle.

The quality of the schedule produced by this algorithm depends primarily on the mechanism used to pick an operation from the *Ready* list. Consider the simplest scenario, where the *Ready* list contains at most one item in each iteration. In this restricted case, the algorithm must generate an optimal schedule. Only one operation can execute in the first cycle. (There must be at least one leaf in $\mathcal{D}$, and our restriction ensures that there is exactly one.) At each subsequent cycle, the algorithm has no choices to make—either *Ready* contains an operation and the algorithm schedules it, or *Ready* is empty and the algorithm schedules nothing to issue in that cycle.

The difficulty arises when, in some cycle, the *Ready* list contains multiple operations. Then, the scheduler must choose among several ready operations and that choice is critical. The algorithm takes the operation with

INTERACTIONS BETWEEN SCHEDULING AND ALLOCATION

Antidependences between operations can limit the scheduler's ability to reorder operations. The scheduler can avoid antidependences by renaming; however, renaming creates a need for the compiler to perform register allocation after scheduling. This example is but one of the interactions between instruction scheduling and register allocation.

The core function of the scheduler is to reorder operations. Since most operations both use and define values, changing the relative order of two operations can change the lifetimes of values. Moving an operation $x \leftarrow y + z$ forward can increase the lifetimes of y and z and decrease the lifetime of x. Symmetrically, moving the operation backward can shrink the lifetimes of y and z while lengthening the lifetime of x. The net impact of reordering operations on register demand depends on detailed information about the lifetimes of the definitions and uses.

In a similar way, register allocation can change the instruction-scheduling problem. The core functions of a register allocator are to rename references and to insert spill code. Both of these functions affect the scheduler's ability to produce fast code. When the allocator maps a large virtual name space to the smaller name space of target-machine registers, it can introduce antidependences that constrain the scheduler. Similarly, when the allocator inserts spill code, it adds operations to the code that must, themselves, be scheduled into instructions.

We know, mathematically, that solving these problems together might produce solutions that cannot be obtained by running the scheduler followed by the allocator or the allocator followed by the scheduler. However, both problems are complex enough that most real-world compilers treat them separately.

the highest priority score. A well-designed priority scheme incorporates lower-priority tie-breaking criteria. The metric suggested earlier, maximum latency-weighted distance to a root, corresponds to always choosing the operation on the critical path for the current cycle in the schedule being constructed. To the extent that the impact of a scheduling priority is predictable, this scheme should provide balanced pursuit of the longest paths.

Scheduling Operations with Variable Delays

The latency of memory operations often depends on the current state of the memory hierarchy. Thus, the latency of a specific load operation might vary from a couple of cycles to hundreds of cycles. Worse yet, a given operation—that is, the instruction in some specific slot in some specific block—may have different latencies on successive executions.

```
for each load operation, l, in the block do
    delay(l) ← 1
for each operation i in 𝒟 do
    let 𝒟ᵢ be the nodes and edges in 𝒟 independent of i
    for each connected component C of 𝒟ᵢ do
        find the maximal number of loads, N, on any path through C
        for each load operation l in C do
            delay(l) ← delay(l) + delay(i) ÷ N
```

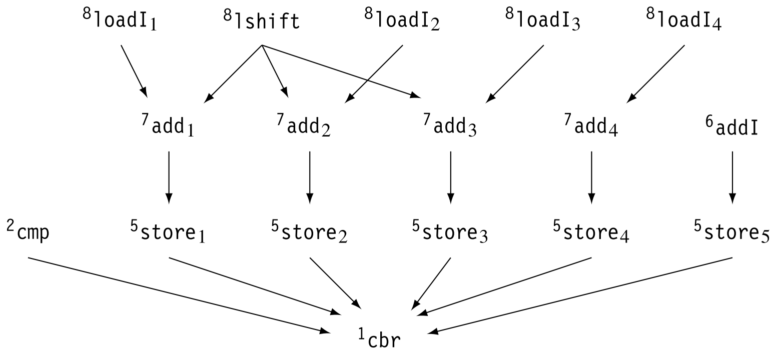
■ **FIGURE 12.7** Computing Delays for Load Operations.

In the absence of detailed knowledge about the addresses being accessed and the state of the memory hierarchy, the scheduler may not have an accurate estimate of the delay. If it assumes the worst case, it risks idling the processor for long periods. If it assumes the best case, it will stall the processor on a cache miss. In practice, the scheduler must seek some middle ground.

The compiler can obtain good results by calculating an individual latency for each load based on the amount of instruction-level parallelism available to cover the load's latency. This approach, called *balanced scheduling*, schedules the load with regard to the code that surrounds it rather than the hardware on which it will execute. It distributes the locally available parallelism across loads in the block. This strategy mitigates the effect of a cache miss by scheduling as much extra delay as possible for each load. It will not slow down execution in the absence of cache misses.

Fig. 12.7 shows the computation of delays for individual loads in a block. The algorithm initializes the delay for each load to one. Next, it considers each operation i in the block's dependence graph, $\mathcal{D}$. It finds the nodes in $\mathcal{D}$ that are independent of i, called $\mathcal{D}_\text{i}$. Conceptually, this task is a reachability problem on $\mathcal{D}$. We can find $\mathcal{D}_\text{i}$ by removing from $\mathcal{D}$ every node that is a transitive predecessor of i or a transitive successor of i, along with any edges associated with those nodes.

The algorithm then finds the connected components of $\mathcal{D}_\text{i}$. For each component C, it finds the maximum number N of loads on any single path through C. N is the number of loads in C that can share operation i's delay, so the algorithm adds $delay(i)/N$ to the delay of each load in C. For a given load l, the operation sums the fractional share of each independent operation i's delay that can be used to cover the latency of l. Using this value as $delay(l)$ produces a schedule that shares the slack time of independent operations evenly across all loads in the block.



■ **FIGURE 12.8** Dependence Graph for a Block from go.

12.3.6 **Forward Versus Backward List Scheduling**

The list-scheduling algorithm, as presented in Fig. 12.6, moves through the dependence graph from its leaves to its roots and creates the schedule from the first cycle in the block to the last. An alternate formulation of the algorithm operates in the opposite direction, moving from roots to leaves and scheduling from last cycle to first cycle. This version of the algorithm is called *backward list scheduling*, and the original version is called *forward list scheduling*.

List scheduling, itself, is not particularly expensive. Thus, some compilers have run the scheduler several times with different combinations of heuristics and kept the best schedule. (The scheduler can reuse the preparatory work—renaming, building the dependence graph, and computing priorities.) In such a scheme, the compiler should consider using both forward and backward scheduling.

In practice, neither forward scheduling nor backward scheduling always wins. The difference between forward and backward list scheduling lies in the order in which the scheduler considers operations. If the schedule depends critically on the careful ordering of some small set of operations, the two directions may produce noticeably different results. If the critical operations occur near the leaves, forward scheduling might consider them together, while backward scheduling would need to work its way through the remainder of the block to reach them. Symmetrically, if the critical operations occur near the roots, backward scheduling might examine them together, while forward scheduling sees them in an order dictated by decisions made starting at the other end of the block.

	Integer	Integer	Memory
1	loadI$_1$	lshift	--
2	loadI$_2$	loadI$_3$	--
3	loadI$_4$	add$_1$	--
4	add$_2$	add$_3$	--
5	add$_4$	addI	store$_1$
6	cmp	--	store$_2$
7	--	--	store$_3$
8	--	--	store$_4$
9	--	--	store$_5$
10	--	--	--
11	--	--	--
12	--	--	--
13	cbr	--	--

(a) Forward Schedule

	Integer	Integer	Memory
1	loadI$_4$	--	--
2	addI	lshift	--
3	add$_4$	loadI$_3$	--
4	add$_3$	loadI$_2$	store$_5$
5	add$_2$	loadI$_1$	store$_4$
6	add$_1$	--	store$_3$
7	--	--	store$_2$
8	--	--	store$_1$
9	--	--	--
10	--	--	--
11	cmp	--	--
12	cbr	--	--

(b) Backward Schedule

■ **FIGURE 12.9** Schedules for the Block from go.

To make this point concrete, consider the example shown in Fig. 12.8. It shows the dependence graph for a basic block found in the SPEC 95 benchmark program go. The compiler added dependences from the store operations to the block-ending branch to ensure that the memory operations complete before the next block begins execution. Superscripts on nodes in the dependence graph give the latency from the node to the end of the block; subscripts differentiate among similar operations. The example assumes the operation latencies shown in the margin.

Opcode	Delay
loadI	1
lshift	1
add	2
addI	1
cmp	1
cbr	1
store	4

This example came to our attention during a study of list scheduling that targeted an ILOC machine with two integer functional units and one unit to perform memory operations. The five store operations take most of the time in the block. To minimize execution time, the schedule must place the stores as early as possible.

Forward list scheduling, using latency to the end of the block for priority, executes the operations in priority order, except for the comparison. It schedules the five operations with rank eight, then the four rank seven operations and the rank six operation. It begins on the operations with rank five, and slides the cmp in alongside the stores, since the cmp is a leaf. If ties are broken arbitrarily by taking left-to-right order, this produces the schedule shown in Fig. 12.9(a). Notice that the memory operations begin in cycle 5, producing a schedule that issues the branch in cycle 13.

WHAT ABOUT OUT-OF-ORDER EXECUTION?

Dynamically scheduled processors, with their support for out-of-order (OOO) execution, can improve on the static schedule that a compiler produces. Does OOO execution eliminate the need for compiler-based scheduling?

To understand this issue, it is important to recognize when an OOO processor can improve on the static schedule. If runtime circumstances are better than the assumptions made by the scheduler, then OOO hardware can issue an operation earlier than its position in the static schedule. This situation can arise at a block boundary if an operand is available before its worst-case time. It can arise with variable-latency operations, such as loads and stores. Because the processor can look at actual runtime addresses, an OOO processor can also disambiguate some load-store dependences that the scheduler cannot.

However, OOO execution does not eliminate the need for the compiler to schedule instructions. Because the lookahead window is finite, bad schedules can defy improvement. For example, a lookahead window of 50 instructions will not let the processor execute a string of 100 integer instructions followed by 100 floating-point instructions in interleaved ⟨integer, floating point⟩ pairs. It should, however, interleave shorter strings of operations. OOO execution helps the compiler by improving good, but nonoptimal, schedules.

Using the same priorities with backward list scheduling, the compiler first places the branch in the last slot of the block. The `cmp` precedes it by one cycle, determined by *delay*(`cmp`). The next operation scheduled is $\texttt{store}_1$ (by the left-to-right tie-breaking rule). It is assigned the issue slot on the memory unit that is four cycles earlier, determined by *delay*(`store`). The scheduler fills in successively earlier slots on the memory unit with the other `store` operations, in order. It begins filling in the integer operations, as they become ready. The first is $\texttt{add}_1$, two cycles before $\texttt{store}_1$. When the algorithm terminates, it has produced the schedule shown in Fig. 12.9(b).

The backward schedule takes one fewer cycle than does the forward schedule. It places the `addI` earlier in the block, which allows $\texttt{store}_5$ to issue in cycle 4—one cycle earlier than the first memory operation in the forward schedule. By considering the problem in a different order, using the same underlying priorities and tie breakers, the backward algorithm finds a different result.

Why does this happen? The forward scheduler places all the rank-eight operations in the schedule before any rank-seven operations. Even though the `addI` operation is a leaf, its lower rank causes the forward scheduler to defer it. By the time the scheduler runs out of rank-eight operations, other rank-seven operations are available. By contrast, the backward scheduler places the `addI` earlier than three of the rank-eight operations—a result that the forward scheduler could not consider.

SECTION REVIEW

List scheduling has been the dominant paradigm that compilers have used for many years. It computes, for each operation, the cycle in which that operation should issue. The algorithm is reasonably efficient; its complexity relates directly to the underlying dependence graph. This greedy heuristic approach, in its forward and backward forms, produces excellent results for single blocks.

The key data structure for instruction scheduling is the dependence graph. It represents the flow of data in the block. It is easily annotated with information about operation-by-operation delays. The dependence graph directly represents both the constraints and the critical paths in the block.

REVIEW QUESTIONS

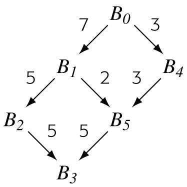
1. The *Ready* list is one of the most heavily accessed data structures in a list scheduler. Compare and contrast the costs of implementing *Ready* as an ordered list versus a priority queue. How does the expected length of the *Ready* list affect the tradeoff?
2. For each of the following tie-breakers, suggest a rationale.
    a. Prefer operations with operands in registers over ones with immediate operands.
    b. Prefer the operation with the most recently defined operands.
    c. Prefer a load before an operation that computes a value.

12.4 REGIONAL SCHEDULING

To extend list scheduling to larger regions, compiler writers use the same kinds of strategies that work with value numbering. They construct multiblock regions that they can treat as a single block and, then, apply the local scheduling algorithm to those regions. As with superlocal value numbering, the compiler must take care on the borders between the multiblock region and the rest of the procedure being compiled. This section examines three schemes that compiler writers have used to improve schedule quality:

superlocal scheduling, trace scheduling, and superblock cloning. Each expands the context to which the compiler applies list scheduling.

12.4.1 Superlocal Scheduling



Example CFG with Execution Frequencies

Recall from Section 8.3 that an extended basic block (EBB) consists of a set of blocks $B_1, B_2, \ldots, B_n$ in which B_1 has multiple predecessors and every other block B_i has exactly one predecessor, some B_j in the EBB. The compiler can identify EBBs in a simple pass over the CFG. The CFG shown in the margin has one large EBB, $\{B_0, B_1, B_2, B_4\}$, and two trivial EBBs, $\{B_3\}$ and $\{B_5\}$. The large EBB has two paths, $\langle B_0, B_1, B_2\rangle$, and $\langle B_0, B_4\rangle$, The paths share B_0 as a common prefix.

Both (B_0, B_4) and (B_1, B_5) are premature exits relative to $\langle B_0, B_1, B_2\rangle$.

To obtain a larger context, the scheduler can treat paths in an EBB, such as $\langle B_0, B_1, B_2\rangle$, as if they are single blocks, provided that it accounts for the shared path prefixes and for any premature exits from those paths. This approach lets the compiler apply list scheduling to longer sequences of operations. The effect is to increase the fraction of code that is scheduled together, which may improve execution times.

Compensation code
code inserted into a block B_i to counteract the effects of cross-block code motion along a path that does not include B_i

To see how shared prefixes and premature exits complicate list scheduling, consider the ways that the scheduler might move operations across block boundaries in the path $\langle B_0, B_1, B_2\rangle$ in the example. Such code motion may require the scheduler to insert *compensation code* to maintain correctness.

- The compiler can move an operation forward—that is, later on the path. For example, it might move an operation i from B_0 into B_1. While that decision might speed execution along the path $\langle B_0, B_1, B_2\rangle$, it removes i from the path $\langle B_0, B_4\rangle$. Unless i is dead along $\langle B_0, B_4\rangle$, the scheduler must correct this situation.

  To compensate, the scheduler can insert a copy of i at the head of B_4. If it was legal to move i out of B_0, it is legal to place it at the head of B_4. A dependence that prevents placement of i at the head of B_4 would also prevent its placement in B_1. The copy of i in B_4 does not lengthen the path $\langle B_0, B_4\rangle$ but it does increase the overall code size.

Because $\langle B_0, B_4\rangle$ is an EBB path, we know B_4 has only one predecessor.

- The compiler can move an operation backward—that is, earlier on the path. For example, it might move an operation j from B_1 to B_0. While that decision might speed execution along the path $\langle B_0, B_1, B_2\rangle$, it adds j to the path $\langle B_0, B_4\rangle$. That placement lengthens the path $\langle B_0, B_4\rangle$. It may also change the values seen in B_4.

  If j kills some value used in B_4, the scheduler must rewrite the code to make the value available at the head of B_4. It could insert code at the head of B_4 to recompute the value; renaming might accomplish the same effect. In either case, the presence of both j and compensation code along the path $\langle B_0, B_4 \rangle$ will lengthen that path.

The mechanics of superlocal scheduling are straightforward. The compiler selects an EBB. It performs renaming over the region, if necessary. Next, it schedules each path through the EBB, in order of decreasing execution frequency. It builds a dependence graph for the entire path and schedules it ignoring premature exits and recording all cross-block placements. A postpass inserts compensation code. It repeats this process until it has scheduled every block in the EBB. Each block is scheduled exactly once.

When the scheduler processes a path with an already-scheduled prefix, it leaves that prefix intact. The effect is to prioritize the schedules of hotter paths over those of cooler paths. If the estimates of future execution frequencies are accurate, this approach may improve overall execution times.

In the example, the scheduler might process $\langle B_0, B_1, B_2 \rangle$. Next, it would schedule B_4 relative to the schedule for B_0. Finally, it would schedule the trivial EBBs, B_3 and B_5, as singleton blocks.

Mitigating Compensation Code

The scheduler can take steps to limit the impact of compensation code. It can use live information to avoid some of the compensation code suggested by forward motion. If the result of the moved operation is not live on entry to the off-path block, no compensation code is needed on that path. It can avoid compensation code introduced by backward motion if it prohibits backward motion into a block b of any operation that defines a name $n \in \text{LIVEOUT}(b)$. This restriction can limit the scheduler's ability to improve the code, but it avoids lengthening other paths.

12.4.2 Trace Scheduling

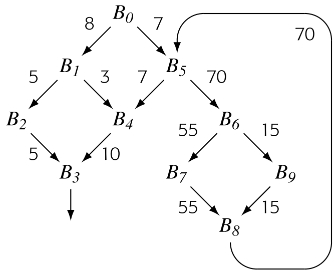
Trace scheduling extends the approach taken in superlocal scheduling beyond a path through a single EBB. Trace scheduling constructs maximal-length acyclic paths, or *traces*, through the CFG and applies the list-scheduling algorithm to those traces. Like superlocal scheduling, this approach schedules each block once. Like superlocal scheduling, it may need to insert compensation code. The algorithm consists of two distinct phases: trace construction and scheduling.

Trace
an acyclic path through the CFG selected using profile information

The algorithm follows the same basic scheme as superlocal scheduling. It identifies a trace and schedules that trace—keeping track of places where the new schedule will need compensation code. It schedules the hottest paths first, so that the shorter and more constrained schedules execute less often.

```
trace ← eligible edge e with highest frequency count
x ← hottest eligible edge entering source(trace)
    or leaving sink(trace)
while (x is nonnull) do
    add x to the appropriate end of the trace
    x ← hottest eligible edge entering source(trace)
        or leaving sink(trace)
```

(a) The Trace Construction Algorithm



(b) A More Complex Example

■ **FIGURE 12.10** Trace Construction Algorithm.

Trace Construction

Fig. 12.10(a) shows the algorithm to construct a single trace. It uses a greedy heuristic to find the hottest acyclic path through the CFG. It begins with a trace that consists of the hottest remaining edge in the graph—the one with highest execution frequency—and repeatedly adds edges to either the start or the end of the trace. At each step, it takes the hottest edge among the available choices—those that would extend either the start or the end of the trace. When no such edge remains, the trace is finished.

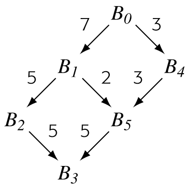
Recall that an edge (x, y) is a *loop-closing* edge in the CFG if $y \in \text{DOM}(x)$.

The algorithm excludes some edges. It considers an edge *ineligible* for use in the current trace if either the edge closes a loop or both its source and sink already appear in the current trace or an earlier one.

The first restriction prevents the scheduler from moving code out of a loop. We assume that the optimizer already performed loop-invariant code motion (see Section 10.3.1) and that the scheduler should not insert compensation code that splits a loop-closing branch.

To find successive traces, the compiler removes the on-trace edges from the graph and invokes the trace construction algorithm on the remaining graph. It repeats this process until no eligible edges remain.

Examples



Example CFG with Execution Frequencies

Consider the CFG shown in the margin. The algorithm starts with the hottest edge, (B_0, B_1). It extends (B_0, B_1) with (B_1, B_2) and (B_2, B_3), in order. At that point, it cannot extend the trace any further. The first trace is $\langle B_0, B_1, B_2, B_3 \rangle$. It removes those edges from the graph.

For the second trace, the algorithm starts with (B_5, B_3). It extends the trace with (B_4, B_5) and (B_0, B_4). It cannot extend the trace further. The second trace is $\langle B_0, B_4, B_5, B_3\rangle$. It removes those edges. The only remaining edge, (B_1, B_5), is ineligible.

Fig. 12.10(b) shows a more complex CFG. Its hottest edge is (B_5, B_6). It extends the trace with, in order, (B_6, B_7), (B_7, B_8), and (B_0, B_5), to form the trace $\langle B_0, B_5, B_6, B_7, B_8\rangle$.

Note that (B_8, B_5) is a loop-closing edge and, therefore, ineligible.

The second trace begins with a tie for the hottest remaining edge. It chooses one of (B_6, B_9) or (B_9, B_8). It uses the other edge to extend the trace, to form $\langle B_6, B_9, B_8\rangle$. At that point, no eligible edge remains to extend the second trace.

The third trace begins with (B_4, B_3). The algorithm adds (B_5, B_4) to form $\langle B_5, B_4, B_3\rangle$. No eligible edge remains to extend this trace.

The fourth trace begins with (B_0, B_1). The algorithm extends the trace with (B_1, B_2) and (B_2, B_3). At that point, no eligible edges remain. The fourth trace is $\langle B_0, B_1, B_2, B_3\rangle$. The only edge left in the graph is (B_1, B_4), which runs between two traces and is, therefore, ineligible.

Scheduling

Given a trace, the scheduler applies the list-scheduling algorithm to the entire trace, in the same way that superlocal scheduling does. With a trace, one additional opportunity for compensation code occurs; the trace may have interim entry points—blocks in mid-trace that have multiple predecessors, such as B_4 in Fig. 12.10(b).

- Forward motion of an operation *i* on the trace across an interim entry point may add *i* to the off-trace path. If *i* redefines a value that is also live across the interim entry, some combination of renaming or recomputation may be necessary. The alternative is to either prohibit forward motion across the interim entry or to use cloning to avoid this situation (see Section 12.4.3).
- Backward motion of an operation *i* across an interim entry point may force the compiler to add *i* to the off-trace path. The change is straightforward, since *i* already occurred on the off-trace path. The scheduler must already correct for any name conflict introduced by the on-trace backward motion; the off-trace compensation code can simply define the same name.

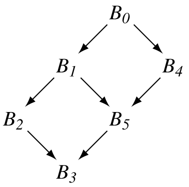
If the interim entry comes from a previously scheduled block, the scheduler may need to split the entering edge, unless the specific situation can be handled with renaming. The alternative is to prohibit code motion across interim entries if it forces compensation code into an already scheduled trace.

To schedule an entire procedure, the scheduler first performs renaming over the entire procedure. Next, it repeatedly discovers a new trace and schedules it, until it cannot construct another trace. To schedule an individual trace, it builds a dependence graph for the trace, computes priorities, and performs list scheduling on the trace. The process continues until all the blocks have been scheduled.

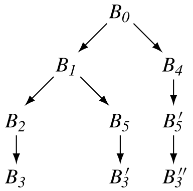
Superlocal scheduling can be considered a degenerate case of trace scheduling in which interim entries to the trace are prohibited.

12.4.3 Cloning for Context

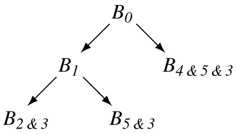
Join points in the control-flow graph can limit the opportunities for either superlocal scheduling or trace scheduling. To improve the results, the compiler can clone blocks to create longer join-free paths. Superblock cloning has exactly this effect (see Section 10.6.1). For superlocal scheduling, it increases the size of the EBBs and the length of some of the paths through the EBBs. For trace scheduling, it avoids the complications caused by interim entry points in the trace. In either case, cloning also eliminates some of the branches and jumps in the EBBs.



Example CFG



Example After Cloning



Example After Combining Blocks

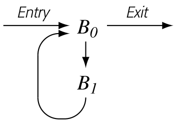
For the example CFG shown in the margin, cloning would produce the CFG labeled “Example After Cloning.” Block B_5 has been cloned to create separate instances for the path from B_1 and the path from B_4. Similarly, B_3 has been cloned twice to create a unique instance for each path that enters it. Taken together, these actions eliminate all join points in the CFG.

At this point, the compiler can combine some of the blocks, to produce the final CFG, “Example After Combining Blocks.” This step eliminates some control-flow operations, such as the jump from B_4 to B_5'.

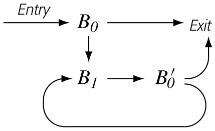
Now, the entire graph forms one single EBB. The compiler can select the hottest path, say $\langle B_0, B_1, B_{2\,\&\,3}\rangle$ and schedule it. It can then use $\langle B_0, B_1\rangle$ as a prefix to schedule $B_{5\,\&\,3}$. Finally, it can use B_0 as a prefix to schedule $B_{4\,\&\,5\,\&\,3}$. Cloning the CFG eliminates most of the interference between these distinct paths.

Contrast this result with the result of superlocal scheduling on the original CFG. The superlocal algorithm scheduled B_4 with respect to B_0, as happens with the cloned graph. However, the superlocal algorithm scheduled both B_3 and B_5 with no prior context. In the cloned graph, each is duplicated and scheduled to a set of specific prefixes. The price of this specialization is increased code size.

Tail-recursive programs can also benefit from cloning. Recall from Sections 7.5.2 and 10.4.1 that a program is tail recursive if its last action is a tail call—a recursive self-invocation. When the compiler detects a tail call, it can convert the call into a jump back to the procedure's entry. As shown in the margin, cloning blocks in the tail-recursive procedure may create a code shape that gives the scheduler a longer block and a more restricted context with which to work.



Tail Call After
Tail-Call Optimization



Tail Call After
Cloning and Optimization

The first diagram shown in the margin shows the abstracted CFG graph for a tail-recursive routine, after the tail call has been optimized. Block B_0 is entered along two paths, the path from the procedure entry and the path from B_1. The scheduler must use worst-case assumptions about what precedes B_0. By cloning B_0 as shown in the lower drawing, the compiler can make control enter B_0' along only one edge, which may improve the results of regional scheduling. To further simplify the situation, the compiler might coalesce B_0' onto the end of B_1, creating a single-block loop body. The resulting loop can be scheduled with either a local scheduler or a loop scheduler, as appropriate.

SECTION REVIEW

Regional scheduling techniques build longer segments of straight-line code to which they apply list scheduling. The infrastructure of regional scheduling simply provides more context and more operations to the list scheduler, in an attempt to provide that scheduler with more freedom and more opportunities. The quality of the code that the regional scheduler produces is, to some extent, determined by the quality of the underlying scheduler.

Superlocal scheduling, trace scheduling, and superblock cloning before superlocal scheduling can each introduce compensation code. The compensation code takes additional space. It may introduce additional operations along some of the paths. However, experience has shown that the benefits of regional scheduling often outweight the costs.

REVIEW QUESTIONS

1. In regional scheduling, cross-block code motion can necessitate the insertion of compensation code. How can the compiler identify situations in which moving an operation across a block boundary will not require compensation code?
2. Both trace scheduling and cloning try to improve on the results of superlocal scheduling. Compare and contrast these approaches and their expected results.

12.5 ADVANCED TOPICS

Compiler optimization has, since the first FORTRAN compiler, focused on improving code in loops. The reason is simple: code inside loops executes more frequently than code outside of loops. This observation has led to the development of specialized scheduling techniques that attempt to decrease the total running time of a loop. These loop-schedulers can create better schedules than a regional scheduler for one simple reason: they account for the flow of values across the entire loop, including the loop-closing branch.

Loop scheduling comes into play when the default scheduler cannot produce compact and efficient code for a loop. If the scheduled loop has no unused issue slots, loop scheduling is unlikely to help. If, on the other hand, it has a significant number of empty issue slots, then loop scheduling may increase functional unit utilization and decrease the loop's overall running time.

Initiation interval
A loop's *initiation interval* is the lag, in cycles, between the start of the ith iteration and the start of the $(i+1)$st iteration.

Software pipelining schedules a loop by mimicking the behavior of a hardware pipeline. It overlaps the executions of successive iterations of the loop so that operations from two or more iterations execute concurrently. The pipelined loop decreases the lag between the start of the ith and $(i+1)$st iterations—the *initiation interval* of the loop. A smaller initiation interval leads to faster overall execution for the entire loop.

12.5.1 The Strategy Behind Software Pipelining

The speed advantage of software pipelining is simple. For a nonpipelined loop, the initiation interval is simply the number of cycles, c, that it takes to execute one iteration of the loop. If a loop executes n iterations, the total execution time of the loop is $n \cdot c$.

After software pipelining, the loop starts the $(i+1)$st iteration before the ith iteration has finished. This strategy produces a smaller initiation interval, ii, which reduces the overall execution time of the loop.

The pipelined loop starts a new iteration every ii cycles. The first $n-1$ iterations each incur a cost of just ii cycles. The rest is overlapped with later iterations. The final iteration incurs the full cost, c, because the final part of the loop executes on its own. Thus, the total cost of the pipelined loop is $(n-1) \cdot ii + c$ cycles. We can guarantee, by construction, that $ii \leq c$.

Loop kernel
The central portion of a software pipelined loop, the *kernel*, executes most of the loop's iterations in an interleaved fashion.

From a code shape perspective, the transformed loop consists of a pipelined *kernel* that performs the steady-state computation, along with a prolog and an epilog to handle the initialization and finalization of the loop. The combined effect is analogous to that of a hardware pipeline, which executes multiple distinct operations concurrently.

Cycle	Functional Unit 0	Comments
−4	loadI @x ⇒ $r_{@x}$	Set up the loop
−3	loadI @y ⇒ $r_{@y}$	with initial loads
−2	loadI @z ⇒ $r_{@z}$	
−1	addI $r_{@x}$, 796 ⇒ r_{ub}	
1	L_1: loadAO r_{arp}, $r_{@x}$ ⇒ r_x	Get x[i] & y[i]
2	loadAO r_{arp}, $r_{@y}$ ⇒ r_y	
3	addI $r_{@x}$, 4 ⇒ $r_{@x}$	Bump the pointers
4	addI $r_{@y}$, 4 ⇒ $r_{@y}$	in shadow of loads
5	mult r_x, r_y ⇒ r_z	The actual work
6	cmp_LE $r_{@x}$, r_{ub} ⇒ r_{cc}	Shadow of mult
7	storeAO r_z ⇒ r_{arp}, $r_{@z}$	Save the result
8	addI $r_{@z}$, 4 ⇒ $r_{@z}$	Bump z's pointer
9	cbr r_{cc} → L_1, L_2	Loop-closing branch
	L_2: ...	

■ **FIGURE 12.11** Example Loop, Local Scheduler, One Functional Unit.

When the pipelined loop executes, the prolog fills the pipeline. If the kernel executes operations from three iterations of the original loop, then each kernel iteration processes roughly one-third of each active iteration of the original loop. To start execution, the prolog must perform enough work to prepare for the last third of iteration 1, the second third of iteration 2, and the first third of iteration 3.

After the loop kernel completes, the epilog completes the final iterations, which empty the pipeline. In the hypothetical example, the epilog must execute the last third of the second-to-last iteration and the last two-thirds of the final one. The prolog and epilog increase the loop's overall code size.

To make this discussion concrete, consider the following loop in C:

```
for (i=0; i < 200; i++)
    z[i] = x[i] * y[i];
```

Assume that x, y, and z all have lower bounds of zero. Fig. 12.11 shows the ILOC code that a compiler might generate for the loop, after optimization and local scheduling. Both operator strength reduction and linear function test replacement have been applied (see Section 10.7.2), so the address expressions for x, y, and z are updated with addI operations and the end of loop test has been rewritten in terms of the offset in x, which eliminates the need for i.

Cycle		Functional Unit 0			Functional Unit 1		
−2		loadI	@x	⇒ $r_{@x}$	loadI	@y	⇒ $r_{@y}$
−1		loadI	@z	⇒ $r_{@z}$	addI	$r_{@x}$, 796	⇒ r_{ub}
1	L_1:	loadAO	r_{arp}, $r_{@x}$	⇒ r_x	nop		
2		loadAO	r_{arp}, $r_{@y}$	⇒ r_y	addI	$r_{@x}$, 4	⇒ $r_{@x}$
3		addI	$r_{@y}$, 4	⇒ $r_{@y}$	mult	r_x, r_y	⇒ r_z
4		cmp_LE	$r_{@x}$, r_{ub}	⇒ r_{cc}	nop		
5		storeAO	r_z	⇒ r_{arp}, $r_{@z}$	addI	$r_{@z}$, 4	⇒ $r_{@z}$
6			*stall on* r_z		cbr	r_{cc}	→ L_1, L_2
7		... *start of next iteration* ...					

■ **FIGURE 12.12** Example Loop, Local Scheduler, Two Functional Units.

Opcode	Latency
loadAO	3
storeAO	3
loadI	1
addI	1
mult	2
cmp_LE	1
cbr	1

Latencies Used in the Example

The code in Fig. 12.11 has been scheduled for a machine with one functional unit. Operation latencies are shown in the margin. The first column in the figure shows the cycle in which each operation issues, normalized to the start of the loop (at label L_1).

The preloop code initializes a pointer for each array ($r_{@x}$, $r_{@y}$, and $r_{@z}$). It computes an upper bound for the range of $r_{@x}$ into r_{ub}; the end-of-loop test uses r_{ub}. The loop body loads x and y, performs the multiply, and stores the result into z. The schedule uses all of the issue slots in the shadow of long-latency operations. During the load latencies, it updates $r_{@x}$ and $r_{@y}$. The comparison executes during the multiply. It fills the slots after the store with the update of $r_{@z}$ and the branch. This produces a tight schedule for a one-functional-unit machine.

Consider what happens to the same code on a two-functional unit, superscalar processor with the same latencies. Assume that loads and stores must execute on unit 0, that functional units stall when an operation issues before its operands are ready, and that the processor cannot issue operations to a stalled unit. Fig. 12.12 shows the execution trace of the loop's first iteration. The mult in cycle 3 stalls because neither r_x nor r_y is ready. It stalls in cycle 4 waiting for r_y, begins executing again in cycle 5, and produces r_z at the end of cycle 6. The storeAO must stall until the start of cycle 7. Assuming that the hardware can tell that $r_{@z}$ contains an address that is distinct from $r_{@x}$ and $r_{@y}$, the processor can issue the first loadAO for the second iteration in cycle 7. If not, then the processor will stall until the store completes.

With two functional units, the code executes more quickly. The preloop is down to two cycles, from four. The initiation interval is down to six cycles, from nine. The critical path executes as quickly as we can expect; the

	Cycle	Functional Unit 0	Functional Unit 1
Prolog	-6	loadI @x ⇒ $r_{@x}$	loadI @y ⇒ $r_{@y}$
	-5	loadAO r_{arp}, $r_{@x}$ ⇒ r_x	addI $r_{@x}$, 796 ⇒ r_{ub}
	-4	loadAO r_{arp}, $r_{@y}$ ⇒ r_y	addI $r_{@x}$, 4 ⇒ r_x
	-3	loadI @z ⇒ $r_{@z}$	addI $r_{@y}$, 4 ⇒ r_y
	-2	nop	nop
	-1	mult r_x, r_y ⇒ r_z	nop
Kernel	1	L_1: loadAO r_{arp}, $r_{@x}$ ⇒ r_x	addI $r_{@x}$, 4 ⇒ $r_{@x}$
	2	loadAO r_{arp}, $r_{@y}$ ⇒ r_y	addI $r_{@y}$, 4 ⇒ $r_{@y}$
	3	cmp_LE $r_{@x}$, r_{ub} ⇒ r_{cc}	nop
	4	storeAO r_z ⇒ r_{arp}, $r_{@z}$	addI $r_{@z}$, 4 ⇒ $r_{@z}$
	5	cbr r_{cc} → L_1, L_2	mult r_x, r_y ⇒ r_z
Epilog	+1	L_2: nop	nop
	+2	storeAO r_z ⇒ r_{arp}, $r_{@z}$	nop
	+3	...	...

■ **FIGURE 12.13** Example Loop, Software Pipelined, Two Functional Units.

multiply issues before r_y is available and executes as soon as possible. The store proceeds as soon as r_z is available. Some issue slots are wasted (unit 0 in cycle 6 and unit 1 in cycles 1 and 4).

Reordering the linear code can change the execution schedule. For example, moving the update of $r_{@x}$ in front of the load from $r_{@y}$ allows the processor to issue the updates of $r_{@x}$ and $r_{@y}$ in the same cycles as the loads from those registers. This lets some of the operations issue earlier in the schedule, but it does nothing to speed up the critical path. The net result is the same—a six-cycle loop.

Pipelining the code can further reduce the initiation interval, as shown in Fig. 12.13. The top portion, above the dotted line, contains the loop's prolog. The central portion contains its five-cycle kernel. The bottom portion, below the second dotted line, contains the loop's epilog. The next subsection presents the algorithm that generated the kernel for this pipelined loop.

12.5.2 An Algorithm for Software Pipelining

To pipeline the loop, the scheduler first constructs an estimate of the minimum initiation interval that the loop needs. Next, it tries to schedule the loop into a kernel of that length; if that process fails, it increases the kernel size by one and tries again. This process halts; in the worst case, the scheduler reaches the initiation interval of the original, nonpipelined loop. Once the scheduler has the kernel, it generates a prolog and epilog to match.

Estimating Kernel Size

To estimate kernel size, the loop scheduler can compute lower bounds on the number of cycles that the kernel must contain.

- The first lower bound arises from a simple observation: the schedule must issue every operation in the loop body. The number of cycles required to issue all the operations is bounded by:

  $RC = max_u(\lceil O_u / N_u \rceil)$

  where u varies over all functional unit types, O_u is the number of operations of type u in the loop and N_u is the number of functional units of type u. RC represents the resource constraint.

Recurrence
a loop-based computation that creates a cycle in the dependence graph

A recurrence must span multiple iterations.

- The second lower bound arises from another simple observation: the initiation interval must be large enough to let each recurrence in the loop complete. The scheduler can compute the lower bound from recurrence lengths as follows:

  $DC = max_r(\lceil d_r / k_r \rceil)$

  where r ranges over all recurrences in the loop body, d_r is the cumulative *delay* around recurrence r, and k_r is the number of iterations that r spans. DC is the dependence constraint.

These two estimates, RC and DC, are lower bounds because the scheduler may not find a schedule that fits within that number of cycles.

The scheduler can use $ii = max(RC, DC)$ as its first estimate for the initiation interval. It tries to schedule the loop into ii. If that attempt fails, it increments ii and tries again. We know that it will succeed for some ii in the range $max(RC, DC) \leq ii \leq c$, where c is the length of the schedule found by the local scheduler.

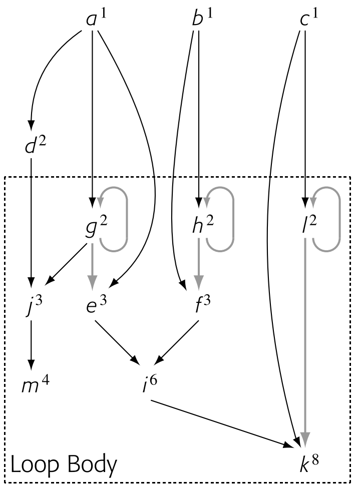
To compute RC, the scheduler must account for two different estimates. The loop has a total of nine operations that are scheduled on two functional units, producing a value for RC of $\lceil 9/2 \rceil = 5$. It has three memory operations that are restricted to unit zero; these operations produce a value for RC of $\lceil 3/1 \rceil = 3$. Thus, the RC constraint for the loop is $max(5, 3) = 5$.

The computation of DC looks at the dependence graph. It contains three recurrences, one for each of $\texttt{r}_{\texttt{@x}}$, $\texttt{r}_{\texttt{@y}}$, and $\texttt{r}_{\texttt{@z}}$. All three have a cumulative delay of one and span one iteration, so $DC = \lceil 1/1 \rceil = 1$.

With $RC = 5$ and $DC = 1$, $ii = max(RC, DC) = 5$.

```
a:       loadI   @x         ⇒ r@x
b:       loadI   @y         ⇒ r@y
c:       loadI   @z         ⇒ r@z
d:       addI    r@x, 796   ⇒ rub

e:   L1: loadAO  rarp, r@x  ⇒ rx
f:       loadAO  rarp, r@y  ⇒ ry
g:       addI    r@x, 4     ⇒ r@x
h:       addI    r@y, 4     ⇒ r@y
i:       mult    rx, ry     ⇒ rz
j:       cmp_LE  r@x, rub   ⇒ rcc
k:       storeAO rz         ⇒ rarp, r@z
l:       addI    r@z, 4     ⇒ r@z
m:       cbr     rcc        → L1, L2

n:   L2: ...
```

(a) Original Code for Example Loop



(b) Dependence Graph

■ **FIGURE 12.14** Dependence Graph for the Example Loop.

Scheduling the Kernel

To schedule the kernel, the compiler uses the list scheduler with a fixed-length schedule of *ii* slots. Updates to the scheduling clock, *Cycle* in Fig. 12.6, are performed modulo *ii*. The cycles in the dependence graph introduce two complications for the scheduler to manage.

Modulo scheduling
List scheduling with a cyclic clock is sometimes called *modulo scheduling*.

First, the scheduler must recognize that loop-carried dependences, such as (g, e), (h, f), and (l, k), do not constrain the first iteration of the loop. The first iteration will use values from before the loop, as indicated by the edges from *a*, *b*, and *c* in the dependence graph from Fig. 12.14(b).

Loop-carried dependence
a dependence that represents a value carried along the CFG edge for the loop-closing branch

Second, the loop-carried dependences expose critical antidependences. In the example, *g* both reads and writes $r_{@x}$. For any given value in $r_{@x}$, *g* cannot update $r_{@x}$ before *e* uses the value in $r_{@x}$. Thus, *g* cannot be scheduled before *e*. (They can issue in the same cycle, because *e* reads its arguments at the start of the cycle and *g* writes its result at the end of the cycle.) By similar logic, *h* cannot precede *f* and *l* cannot precede *k*.

Fig. 12.15 shows the steps that the scheduler takes with the example code. It assumes a two functional-unit processor with memory operations restricted to unit zero, and an initiation interval of five.

Cycle 1 The ready list starts with (e, f). Antidependences exclude *g*, *h*, and *l*. The scheduler chooses *e* and places *e* on unit 0. That placement

Cycle	F0	F1
1	*e*	*g*

Cycle 1

Cycle	F0	F1
1	*e*	*g*
2	*f*	*h*

Cycle 2

Cycle	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—

Cycle 3

Cycle	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—
4	—	—

Cycle 4

Cycle	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—
4	—	—
5	*m*	*i*

Cycle 5

Cycle	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—
4	—	—
5	*m*	*i*

Cycles 1, 2, and 3

Cycle	F0	F1
1	*e*	*g*
2	*f*	*h*
3	*j*	—
4	*k*	*l*
5	*m*	*i*

Cycle 4

■ **FIGURE 12.15** Steps in Modulo Scheduling the Example.

satisfies the antidependence on *g*, so it moves *g* onto the ready list. *g* has a longer distance to a root than *f*, so the scheduler places *g* on unit 1, as shown in the first panel.

Cycle 2 Both *f* and *j* are ready. Using distance to a root, the scheduler places *f* on unit 0, which adds *h* to the ready list. The scheduler picks *h* based on distance to a root, and places *h* on unit 1.

Cycle 3 Only *j* is ready. The scheduler places it on unit 0.

Cycle 4 At this point, *m* is ready. However, the scheduler is constrained to place the branch so that control transfers after cycle 5. Thus, the scheduler leaves cycle 4 empty.

Cycle 5 Both *m* and *i* are ready. The scheduler places them in cycle 5.

Cycle 1 When the cycle advances, it wraps to 1. The ready list is empty, so nothing can be scheduled. *k* is still on the active list, so the scheduler bumps the counter and continues.

Cycle 2 Operation *k* is ready. Because it is a `store`, it must execute on unit 0. Unit 0 is busy in cycle 2, so the scheduler bumps the counter.

Cycle 3 Unit 0 is still busy, so the scheduler bumps the counter.

Cycle 4 In cycle 4, unit 0 is free. The scheduler places *k* on unit 0 for cycle 4. That placement satisfies the antidependence and makes *l* ready. The scheduler places *l* on unit 1 in cycle 4.

Cycle	Functional Unit 0	Functional Unit 1
1	L_1: loadAO $r_{arp}, r_{@x} \Rightarrow r_x$	addI $r_{@x}, 4 \Rightarrow r_{@x}$
2	loadAO $r_{arp}, r_{@y} \Rightarrow r_y$	addI $r_{@y}, 4 \Rightarrow r_{@y}$
3	cmp_LE $r_{@x}, r_{ub} \Rightarrow r_{cc}$	nop
4	storeAO $r_z \Rightarrow r_{arp}, r_{@z}$	addI $r_{@z}, 4 \Rightarrow r_{@z}$
5	cbr $r_{cc} \rightarrow L_1, L_2$	mult $r_x, r_y \Rightarrow r_z$

■ **FIGURE 12.16** Final Kernel Schedule for the Pipelined Loop.

The scheduler repeatedly bumps the counter (modulo 5) until both the active list and the ready list are empty. Since neither operation *k* nor operation *l* has any descendants in the dependence graph, both ready and active become empty and the algorithm halts.

Changing the address computation can lead to a shorter kernel, as shown in Exercise 9.

Similarly, use of an autoincrement on $r_{@x}$, $r_{@y}$, and $r_{@z}$ would remove operations from the loop and change the schedule.

Modulo scheduling fails when it does not find an issue slot for some operation. If that happens, the algorithm increments the initiation interval and tries again. The process must halt; when *ii* reaches the original schedule length, it will schedule in a single iteration.

Generating Prolog and Epilog Code

In principle, generating the prolog and epilog code is simple. The key insight, in both cases, is that the compiler can use the dependence graph to guide the process.

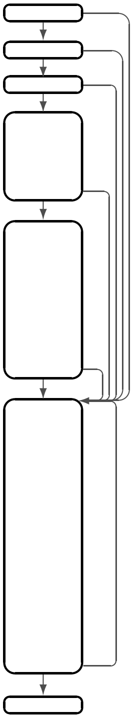
To generate the prolog code, the compiler starts from each upward exposed use in the loop and follows the dependence graph in a backward scheduling phase. For each upward exposed use, it must emit the chain of operations that computes the necessary value, properly scheduled to cover their latencies. To generate the epilog, the compiler starts from each downward exposed use in the loop and follows the dependence graph in a forward scheduling phase.

The kernel of our pipelined loop, shown in Fig. 12.16, has upward exposed uses for $r_{@x}$, $r_{@y}$, $r_{@z}$, r_{ub}, and r_z. The original loop had the first four uses. The exposed reference to r_z arose because the scheduler moved the storeAO operation from the kernel's *i*th iteration to its $(i+1)$st iteration.

Thus, the prolog still needs the operations that define $r_{@x}$, $r_{@y}$, $r_{@z}$, and r_{ub}. In addition, it must compute the first value of r_z and bump the addresses in $r_{@x}$ and $r_{@y}$ so that the first iteration of the kernel reads the second values of x and y. These constraints produce the prolog shown in Fig. 12.13.

```
            B     .S1   LOOP         ; branch to loop
            B     .S1   LOOP         ; branch to loop
            B     .S1   LOOP         ; branch to loop
            B     .S1   LOOP         ; branch to loop
||          ZERO  .L1   A2           ; zero A side product
||          ZERO  .L2   B2           ; zero B side product
            B     .S1   LOOP         ; branch to loop
||          ZERO  .L1   A3           ; zero A side accumulator
||          ZERO  .L2   B3           ; zero B side accumulator
||          ZERO  .D1   A1           ; zero A side load value
||          ZERO  .D2   B1           ; zero B side load value
-------------------------------------------------------------
LOOP:       LDW   .D1   *A4++, A1    ; load a[i] & a[i+1]
||          LDW   .D2   *B4++, B1    ; load b[i] & b[i+1]
||          MPY   .M1X  A1, B1, A2   ; a[i] * b[i]
||          MPYH  .M2X  A1, B1, B2   ; a[i+1] * b[i+1]
||          ADD   .L1   A2, A3, A3   ; ca += a[i] * b[i]
||          ADD   .L2   B2, B3, B3   ; cb += a[i+1] * b[i+1]
|| [B0]     SUB   .S2   B0, 1, B0    ; decrement loop counter
|| [B0]     B     .S1   LOOP         ; branch to loop
-------------------------------------------------------------
            ADD   .L1X  A3, B3, A3   ; c = ca + cb
```

(a) TMS320C6x Assembly Code for Example



(b) CFG for the Loop

■ **FIGURE 12.17** Software Pipelining in a VLIW DSP Compiler.

In this case, the epilog is simple. The kernel executes the entire final iteration, except for the storeA0. Thus, the epilog contains that operation, placed to ensure that the multiply completes before the store issues.

12.5.3 A Final Example

```
c = 0;
for (i=0;i<n;i++) {
    c = c + a[i] * b[i];
}
```

C Loop for Example

Compilers for high-performance DSP architectures have used software pipelining to take advantage of their wide instructions. As an example, consider the simple loop shown in the margin. In 2002, a colleague at Texas Instruments sent us the code that their compiler generated to execute this loop on the TMS320C6x processors. Fig. 12.17(a) shows the final code, which has been optimized so that it has a single instruction, 8 operation, kernel.

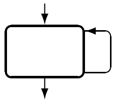
C6x processors can bundle up to eight operations in a cycle. The || sequence at the start of a line indicates that the operation goes into the same instruction bundle as the operation on the previous line. The processor has two register banks, the A and B banks. The codes that begin with a period, such as .S1, dictate where in the VLIW instruction this operation fits. Finally, the doubleword load (LDW) reads two numbers from memory; the multiply low (MPY) and multiply high (MPYH) multiply the low-order and high-order numbers, respectively.

The compiler pipelined the loop and fit it into a single VLIW instruction—the operations between the dashed lines in panel (a). The branch has a five-cycle latency; operations can execute in the delay slots of a branch. Thus, the loop prolog code consists of five instructions, each containing a branch. The last two instructions in the prolog also zero the various registers used in the loop, to create the correct initial conditions.

Several details are missing in the code fragment. We assume that B0, A4, and B4 are initialized earlier.

From a software pipelining perspective, the compiler found the minimal initiation interval—it cannot do better than one cycle. The control flow, shown in panel (b), is counter-intuitive. The prolog fills the pipeline with pending branches, so that every iteration of the kernel executes in the delay slot of a branch issued several cycles earlier. We would not expect a typical human programmer to discover this code shape.

The example arose as part of a study on how to derive accurate representations, including CFGs, from optimized assembly code. Needless to say, the CFG in Fig. 12.17(b) obfuscates the original loop structure. Algorithmically deriving the CFG in panel (b) is complicated. Deriving the CFG for the original loop from the optimized assembly code appears to be much harder. The original CFG appears in the margin.


CFG for the Original Loop

12.6 SUMMARY AND PERSPECTIVE

To obtain reasonable performance on a modern processor, the compiler must perform instruction scheduling. Most compilers use some form of list scheduling. The algorithm is easily adapted and parameterized by changing priority schemes, tie-breaking rules, and even the direction of scheduling. List scheduling is robust, in the sense that it produces good results across a wide variety of codes. In practice, it often finds a time-optimal schedule.

Variations on list scheduling that work over larger regions address problems that arise, at least in part, from increased processor complexity. Regional and loop scheduling techniques are responses to the increase in both the number of pipelines and their individual latencies. As machines have become more complex, schedulers need more context to discover enough instruction-level parallelism to keep the machines busy. Trace scheduling was developed for VLIW architectures, which provide many functional units. Software pipelining provides a way of increasing the number of operations issued per cycle and decreasing total time for executing a loop.

CHAPTER NOTES

Scheduling problems arise in many domains, ranging from construction, through industrial production, through package delivery to spaceflight payload planning. A rich literature has developed on scheduling, including many specialized variants of the problem. Instruction scheduling has been studied as a distinct problem since the 1960s.

Algorithms that guarantee optimal schedules exist for simple situations. For example, on a machine with one functional unit and uniform operation latencies, the Sethi-Ullman labelling algorithm creates an optimal schedule for an expression tree [321]. It can be adapted to produce good code for expression DAGs. Fischer and Proebsting built on the Sethi-Ullman algorithm to derive an algorithm that produces optimal or near optimal results for small memory latencies [299]. Unfortunately, it has trouble as latencies rise or the number of functional units grows.

Much of the scheduling literature deals with variants on the list-scheduling algorithm. Landskov et al. is often cited as the definitive work on list scheduling [248], but the algorithm goes back, at least, to Heller in 1961 [198]. Other papers that build on list scheduling include Bernstein and Rodeh [42], Gibbons and Muchnick [169], and Hennessy and Gross [199]. Krishnamurthy et al. provide a high-level survey of the literature for pipelined processors [243,332]. Kerns, Lo, and Eggers developed balanced scheduling as a way to adapt list scheduling to uncertain memory latencies [232,257]. Schielke's RBF algorithm explored the use of randomization and repetition as an alternative to multilayered priority schemes [318]. Motwani et al. used α-β tuning to blend two heuristics: one that decreases register pressure and another that increases ILP [278].

Many authors have described regional scheduling algorithms. The first automated regional technique appears to be Fisher's trace-scheduling algorithm [158,159]. It has been used in several commercial systems [148,259] and numerous research systems [330]. Hwu et al. proposed *superblock* scheduling as an alternative [211]; inside a loop, it clones blocks to avoid join points, as shown in Section 12.4.3.

Click proposed a global scheduling algorithm based on the use of a global value graph [90]; it was used in the original HotSpot Server Compiler for JAVA [288]. Bala and Rubin developed a technique that can efficiently perform aggressive scheduling techniques, such as cross-block code motion and updates to previously scheduled blocks [33]. They applied the ideas in a global scheduler for the Kendall Square Research compiler.

Several authors have proposed techniques to make use of specific hardware features [311,330]. Other approaches that use replication to improve scheduling include Ebcioğlu and Nakatani [147] and Gupta and Soffa [183]. Sweany and Beaty proposed choosing paths based on dominance information [339]; others have looked at various aspects of that approach [115,210,338].

Software pipelining has been explored extensively. Rau and Glaeser introduced the idea in 1981 [304]. Lam developed the scheme for software pipelining presented here [245]; the paper includes a hierarchical scheme for handling control flow inside a loop. Aiken and Nicolau developed a similar approach, called *perfect pipelining* [11] at the same time as Lam's work.

The example for backward versus forward scheduling in Fig. 12.8 was brought to our attention by Philip Schielke [318]. He took it from the SPEC 95 benchmark program go. It captures, concisely, an effect that has caused many compiler writers to include both forward and backward schedulers in their compilers' back ends.

EXERCISES

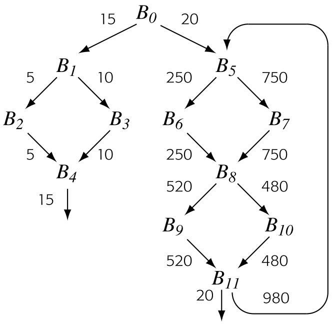
Section 12.3

1. Some operations, such as a register-to-register copy, can execute on almost any functional unit, albeit with a different opcode. Can the scheduler capitalize on these alternatives? Suggest modifications to the basic list-scheduling framework that allow it to use "synonyms" for a basic operation such as a copy.
2. Most modern microprocessors have *delay slots* on some or all branch operations. With a single delay slot, the operation immediately following the branch executes while the branch processes; thus, the ideal slot for scheduling a branch is in the second-to-last cycle of a basic block. (Most processors have a version of the branch that does not execute the delay slot, so that the compiler can avoid generating a nop instruction in an unfilled delay slot.)

   a. How would you adapt the list-scheduling algorithm to improve its ability to "fill" delay slots?

   b. Sketch a postscheduling pass that would fill delay slots.

   c. Propose a creative use for the branch-delay slots that cannot be filled with useful operations.
3. Maintenance of the *Ready* and *Active* lists is a major part of the list scheduling algorithm (see Fig. 12.6). At each cycle, the scheduler must determine which operations in *Active* have completed, and then check



■ **FIGURE 12.18** CFG for Question 7.

each successor of a newly completed operation to see if its operands are now ready.

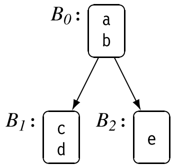
a. Checking, for each successor *s*, whether all of the operands that *s* uses are yet available is an expensive part of list scheduling. Suggest a scheme to reduce the asymptotic cost of those checks.

b. To avoid scanning the entire *Active* list, the compiler writer could implement *Active* with a separate list for each cycle. How many lists would be needed? How would this scheme affect the cost of scheduling?

c. A clever implementation of the multiple list scheme from part (b) could reuse the list. Sketch the code needed to recycle the lists.

Section 12.4

4. In superlocal scheduling, what data structures must the scheduler preserve at the end of a block with multiple successors? Suggest an efficient way to preserve them.



CFG Fragment for Exercise 5

5. Consider the CFG fragment shown in the margin.

   Assume that the scheduler has decided to move c from B_1 into B_0 between a and b. (If that motion is safe, then c does not change a value used in b.) The scheduler must ensure that c also has no impact on e.

   1. If c redefines a value used in e, how might the scheduler use renaming to avoid the impact on e? Which definitions and uses must be renamed?
   2. If the value is used again in some block that follows B_1, how might the scheduler ensure that those uses receive the correct value? Phrase your answer in terms of the LiveOut sets for B_1 and B_2.

6. Working again with the CFG in the margin, assume that the scheduler has decided to move a into block B_2. Under what conditions can it avoid inserting a copy of a into B_1?

Cycle	Functional Unit 0	Comments
−5	addI r_{arp}, @x ⇒ $r_{@x}$	Set up the loop
−4	addI r_{arp}, @y ⇒ $r_{@y}$	with initial loads
−3	addI r_{arp}, @z ⇒ $r_{@z}$	
−2	loadI 0 ⇒ r_{ctr}	
−1	loadI 792 ⇒ r_{ub}	
1	L_1: loadAO r_{ctr}, $r_{@x}$ ⇒ r_x	Get x[i] & y[i]
2	loadAO r_{ctr}, $r_{@y}$ ⇒ r_y	
3	mult r_x, r_y ⇒ r_z	Will stall on r_x & r_y
4	cmp_LE r_{ctr},r_{ub} ⇒ r_{cc}	Shadow of mult
5	storeAO r_z ⇒ r_{ctr}, $r_{@z}$	Save the result
6	addI r_{ctr}, 4 ⇒ r_{ctr}	Bump the offset counter
7	cbr r_{cc} → L_1, L_2	Loop-closing branch
	L_2: ...	

■ **FIGURE 12.19** Code for Question 9.

7. Using the CFG shown in Fig. 12.18.

   a. List the extended basic blocks in the CFG, along with each path through each EBB. In what order should the compiler schedule those paths?

   b. List the traces that the trace construction algorithm will find in the CFG, in the order that they should be scheduled.

   c. Show the CFG that would result from applying superblock cloning to the CFG in Fig. 12.18. Label nodes with their estimated execution counts.

Section 12.5

8. Software pipelining overlaps loop iterations to create an effect that resembles hardware pipelining.

   a. Discuss the impact that software pipelining will have on the demand for registers in the pipelined loop (versus the original loop)?

   b. How can the scheduler use predicated execution to reduce the code-space penalty for software pipelining? (Predicated execution is discussed in Section 7.4.2 on page 352.)

9. The example code in Fig. 12.11 generates a five-cycle software pipelined kernel because it contains nine operations. If the compiler

generated addresses for x, y, and z in a different way, as shown in Fig. 12.19, it could further reduce the operation count in the loop body. This scheme uses one more register than the original code.

a. Compute *RC* and *DC* for this version of the loop.

b. Generate the software pipelined loop body.

c. Generate the prolog and epilog code for your pipelined loop body.

Chapter 13

Register Allocation

ABSTRACT

The code generated by a compiler must make effective use of the limited resources of the target processor. Among the most constrained resources is the set of hardware registers. Register use plays a major role in determining the performance of compiled code. At the same time, register allocation—the process of deciding which values to keep in registers—is a combinatorially hard problem.

Most compilers decouple decisions about register allocation from other optimization decisions. Thus, most compilers include a separate pass for register allocation. This chapter begins with local register allocation, as a way to introduce the problem and the terminology. The bulk of the chapter focuses on global register allocation and assignment via graph coloring. The advanced topics section discusses some of the many variations on that technique that have been explored in research and employed in practice.

KEYWORDS

Register Allocation, Register Spilling, Copy Coalescing, Graph-Coloring Allocators

13.1 INTRODUCTION

Registers are a prominent feature of most processor architectures. Because the processor can access registers faster than it can access memory, register use plays an important role in the runtime performance of compiled code. Register allocation is sufficiently complex that most compilers implement it as a separate pass, either before or after instruction scheduling.

The register allocator determines, at each point in the code, which values will reside in registers and which will reside in memory. Once that decision is made, the allocator must rewrite the code to enforce it, which typically adds load and store operations to move values between memory and specific registers. The allocator might relegate a value to memory because the code contains more live values than the target machine's register set can hold. Alternatively, the allocator might keep a value in memory between uses because it cannot prove that the value can safely reside in a register.

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00019-X

Conceptual Roadmap

A compiler's register allocator takes as input a program that uses some arbitrary number of registers. The allocator transforms the code into an equivalent program that fits into the finite register set of the target machine. It decides, at each point in the code, which values will reside in registers and which will reside in memory. In general, accessing data in registers is faster than accessing it in memory.

Spill
When the allocator moves a value from a register to memory, it is said to *spill* the value.

Restore
When the allocator retrieves a previously spilled value, it is said to *restore* the value.

To transform the code so that it fits into the target machine's register set, the allocator inserts load and store operations that move values, as needed, between registers and memory. These added operations, or "spill code," include loads, stores, and address computations. The allocator tries to minimize the runtime costs of these spills and restores.

As a final complication, register allocation is combinatorially hard. The problems that underlie allocation and assignment are, in their most general forms, NP-complete. Thus, the allocator cannot guarantee optimal solutions in any reasonable amount of time. A good register allocator runs quickly—somewhere between $\mathbf{O}(n)$ and $\mathbf{O}(n^2)$ time, where n is the size of the input program. Thus, a good register allocator computes an effective approximate solution to a hard problem, and does it quickly.

A Few Words About Time

The register allocator runs at compile time to rewrite the almost-translated program from the IR program's name space into the actual registers and memory of the target ISA. Register allocation may be followed by a scheduling pass or a final optimization, such as a peephole optimization pass.

The allocator produces code that executes at runtime. Thus, when the allocator reasons about the cost of various decisions, it makes a compile-time estimate of the expected change in running time of the final code. These estimates are, of necessity, approximations.

A few compiler systems have included description-driven, retargetable register allocators. To reconfigure these systems, the compiler writer builds a description of the target machine at design time; build-time tools then construct a working allocator.

Overview

To simplify the earlier phases of translation, many compilers use an IR in which the name space is not tied to either the address space of the target processor or its register set. To translate the IR code into assembly code for the target machine, these names must be mapped onto the hardware resources of the target machine. Values stored in memory in the IR program must be turned into runtime addresses, using techniques such as those described in Section 7.3. Values kept in virtual registers (VRs) must be mapped into the processor's physical registers (PRs).

Virtual register
a symbolic register name that the compiler uses, before register allocation

We write virtual registers as VR_i, for $i \geq 0$.

Physical register
an actual register on the target processor

We write physical registers as PR_i, for $i \geq 0$.

The underlying memory model of the IR program determines, to a large extent, the register allocator's role (see Section 4.7.1).

- With a register-to-register memory model, the IR uses as many VRs as needed without regard for the size of the PR set. The register allocator must then map the set of VRs onto the set of PRs and insert code to move values between PRs and memory as needed.
- With a memory-to-memory model, the IR keeps all program values in memory, except in the immediate neighborhood of an operation that defines or uses the value. The register allocator can promote a value from memory to a register for a portion of its lifetime to improve performance.

Thus, in a register-to-register model, the input code may not be in a form where it could actually execute on the target computer. The register allocator rewrites that code into a name space and a form where it can execute on the target machine. In the process, the allocator tries to minimize the cost of the new code that it inserts. By contrast, in a memory-to-memory model, all of the data motion between registers and memory is explicit; the code could execute on the target machine. In this situation, allocation becomes an optimization that tries to keep some values in registers longer than the input code did.

This chapter focuses on global register allocation in a compiler with a register-to-register memory model. Section 13.3 examines the issues that arise in a single-block allocator; that local allocator, in turn, helps to motivate the need for global allocation. Section 13.4 explores global register allocation via graph coloring. Finally, Section 13.5 explores variations on the global coloring scheme that have been discussed in the literature and tried in practical compilers.

Graph coloring
an assignment of colors to the nodes of a graph so that two nodes, n_1 and n_2, have different colors if the graph contains the edge (n_1, n_2).

13.2 BACKGROUND

The design and implementation of a register allocator is a complex task. The decisions made in allocator design may depend on decisions made earlier in the compiler. At the same time, design decisions for the allocator may dictate decisions in earlier passes of the compiler. This section introduces some of the issues that arise in allocator design.

13.2.1 A Name Space for Allocation: Live Ranges

At its heart, the register allocator creates a new name space for the code. It receives code written in terms of VRs and memory locations; it rewrites the code in a way that maps those VRs onto both the physical registers and some additional memory locations.

To improve the efficiency of the generated code, the allocator should minimize unneeded data movement, both between registers and memory, and among registers. If the allocator decides to keep some value *x* in a physical register, it should arrange, if possible, for each definition of *x* to target the same PR and for each use of *x* to read that PR. This goal, if achieved, eliminates unneeded register-to-register copy operations; it may also eliminate some stores and loads.

Live range
a closed set of related definitions and uses

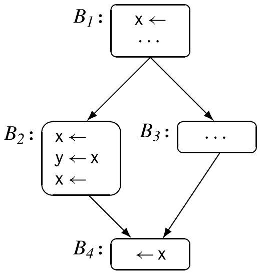
Most allocators use live ranges as values that they consider for placement in a physical register or in memory.

Most register allocators construct a new name space: a name space of *live ranges*. Each live range (LR) corresponds to a single value, within the scope that the allocator examines. The allocator analyzes the flow of values in the code, constructs its set of LRs, and renames the code to use LR names. The details of what constitutes an LR differ across the scope of allocation and between different allocation algorithms.

In a single block, LRs are easy to visualize and understand. Each LR corresponds to a single definition and extends from that definition to the value's last use. Fig. 13.1(a) shows an ILOC fragment that appeared in Chapter 1; panel (b) shows the code renamed into its distinct live ranges. The live ranges are shown as a graph in panel (c). The graph can be summarized as a set of intervals; for example LR_2 is [6,9] and LR_8 is [3,6]. The drawing in panel (c) assumes no overlap between execution of the operations.

We denote LR_8 as starting in operation three because the operation reads its arguments at the start of its execution and writes its result at the end of its execution. Thus, LR_8 is actually defined at the end of the second operation. By convention, we mark a live range as starting with the first operation *after* it has been defined. This treatment makes clear that the two instances of r_1 in addI r_1, 10 ⇒ r_1 are not the same live range, unless other context makes them so.

An LR ends for one of two reasons. An operation may be the name's last use along the current path through the code. Alternatively, the code might redefine the name before its next use, to start a new LR.



Live Ranges in Code with Control Flow

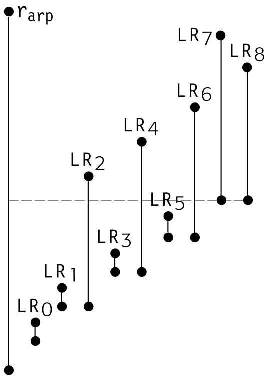
In a CFG with control flow, the situation is more complex, as shown in the margin. Consider the variable x. Its three definitions form two separate and distinct live ranges.

```
 1  loadAI   r_arp, @a ⇒ r_a
 2  loadI    2         ⇒ r_2
 3  loadAI   r_arp, @b ⇒ r_b
 4  loadAI   r_arp, @c ⇒ r_c
 5  loadAI   r_arp, @d ⇒ r_d
 6  mult     r_a, r_2  ⇒ r_a
 7  mult     r_a, r_b  ⇒ r_a
 8  mult     r_a, r_c  ⇒ r_a
 9  mult     r_a, r_d  ⇒ r_a
10  storeAI  r_a ⇒ r_arp, @a
```

(a) Example from Section 1.3.3

```
loadAI   r_arp,@a   ⇒ LR7
loadI    2          ⇒ LR8
loadAI   r_arp,@b   ⇒ LR6
loadAI   r_arp,@c   ⇒ LR4
loadAI   r_arp,@d   ⇒ LR2
mult     LR7,LR8    ⇒ LR5
mult     LR5,LR6    ⇒ LR3
mult     LR3,LR4    ⇒ LR1
mult     LR1,LR2    ⇒ LR0
storeAI  LR0 ⇒ r_arp,@a
```

(b) Code Renamed into LRs



(c) Live Range Spans

■ **FIGURE 13.1** Live Ranges in a Basic Block.

1. The use in B_4 refers to two definitions: the one in B_1 and the one at the bottom of B_2. These three events create the first LR, denoted LR_1. LR_1 spans B_1, B_3, B_4, and the last statement in B_2.
2. The use of x in B_2 refers only to the definition that precedes it in B_2. This pair creates a second LR, denoted LR_2. LR_1 and LR_2 are independent of each other.

With more complex control flow, live ranges can take on more complicated shapes. In the global allocator from Section 13.4, an LR consists of all the definitions that reach some use, plus all of the uses that those definitions reach. This collection of definitions and uses forms a closed set. The interval notation, which works well in a single block, does not capture the complexity of this situation.

Variations on Live Ranges

Different allocation algorithms have defined *live range* in distinct ways. The local allocator described in Section 13.3 treats the entire lifetime of a value in the basic block as a single live range; it uses a maximal-length live range within the block. The global allocator described in Section 13.4 similarly uses a maximal-length live range within the entire procedure.

The live ranges are as long as possible, given the accuracy of the underlying analysis. More precise information about ambiguity might lengthen some live ranges.

Other allocators use different notions of a live range. The linear scan allocators use an approximation of live ranges that overestimates their length but produces an interval representation that leads to a faster allocator. The SSA-based allocators treat each SSA name as a separate live range; they must then translate out of SSA form after allocation. Several allocators have restricted the length of live ranges to conform to features in the control-flow graph, such as loops, to help control spill code placement. Section 13.5.3 describes some of these other formulations.

Code Shape and Live Ranges

The register allocator must understand when a source-code variable can legally reside in a register. If the variable is ambiguous, it can only reside in a register between its creation and the next store operation in the code (see Section 4.7.2). If it is unambiguous and scalar, then the allocator can choose to keep it in a register over a longer period of time.

For the purposes of this discussion, a variable is *scalar* if it is a single value that can fit into a register.

The compiler has two major ways to determine when a value is unambiguous. It can perform static analysis to determine which values are safe to keep in a register; such analysis can range from inexpensive and imprecise through costly and precise. Alternatively, it can encode knowledge of ambiguity into the shape of the code.

If the compiler uses a register-to-register memory model, it can allocate a VR to each unambiguous value. If the VR is live after the return from the procedure that defines it, as with a static value or a call-by-reference parameter, it will also need a memory home. The compiler can save the VR at the necessary points in the code.

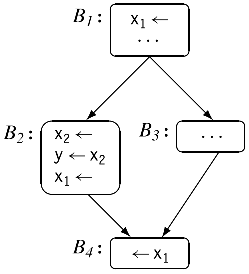
If the IR uses a memory-to-memory model, the allocator will still benefit from knowledge about ambiguity. The compiler should record that information with the symbol table entry for each value.

13.2.2 Interference

Interference
Two live ranges LR$_1$ and LR$_2$ *interfere* if there exists an operation where both are live, and the compiler cannot prove that they have the same value.

The register allocator's most basic function is to determine, for two live ranges, whether or not they can occupy the same register. Two LRs can share a register if they use the same class of register and they are not simultaneously live. If two LRs use the same class of register, but there exists an operation where both LRs are live, then those LRs cannot use the same register, unless the compiler can prove that they have the same value. We say that such live ranges *interfere*.

Two LRs that use physically distinct classes of registers cannot interfere because they do not compete for the same resource. Thus, for example, a floating-point LR cannot interfere with an integer LR on a processor that uses distinct registers for these two kinds of values.



In the example CFG in the margin, the two LRs for x do not interfere; x_2 is only live inside B_2, in a stretch of code where x_1 is dead. Thus, the allocation decisions for x_1 and x_2 are independent. They could share a PR, but there is no inherent reason for the allocator to make that choice.

Global allocators operate by finding interferences and using them to guide the allocation process. The allocator described in Section 13.4 builds a concrete representation of these conflicts, an *interference graph*, and constructs a coloring of the graph to map live ranges onto PRs. Many global allocators follow this paradigm; they vary in the graph's precision and the specific coloring algorithm used.

Interference graph
a graph $G = (N,E)$ that has a node n for each LR and an edge (LR_i,LR_j) if and only if LR_i and LR_j interfere

Finding Interferences

To discover interferences, the compiler first computes live information for the code. Then, it visits each operation in the code and adds interferences. If the operation defines LR_i, the allocator adds an interference to every LR_j that is live at that operation.

The one exception to this rule is a copy operation, $\mathrm{LR}_i \leftarrow \mathrm{LR}_j$ which sets the value of LR_i to the value of LR_j. Because the source and destination LRs have the same value, the copy operation does not create an interference between them. If LR_i and LR_j do not otherwise interfere, they could occupy the same PR.

Interference and Register Pressure

The interference graph provides a quick way to estimate the demand for registers, often called *register pressure*. For a node LR_i in the graph, the *degree* of LR_i, written LR_i°, is the number of neighbors that LR_i has in the graph. If all of LR_i's neighbors are live at the same operation, then $\mathrm{LR}_i^{\circ} + 1$ registers would be needed to keep all of these values in registers. If those values are not all live at the same operation, then the register pressure may be lower than the degree. Maximum degree across all the nodes in the interference graph provides a quick upper bound on the number of registers required to execute the program without any spilling.

Register pressure
a term often used to refer to the demand for registers

Representing Physical Registers

Often, the allocator will include nodes in the interference graph to represent PRs. These nodes allow the compiler to specify both connections to PRs and interferences with PRs. For example, if the code passes LR_i as the second parameter at a call site, the compiler could record that fact with a copy from LR_i to the PR that will hold the second parameter.

Pseudointerference
If the compiler adds an edge between LR_i and PR_j, it must recognize that the edge does not actually contribute to demand for registers.

Some compilers use PRs to control assignment of an LR. To force LR_i into PR_j, the compiler can add a pseudointerference from LR_i to every PR except PR_j. Similarly, to prevent LR_i from using PR_j, the compiler can add an interference between LR_i and PR_j. While this mechanism works, it can become cumbersome. The mechanism for handling overlapping register classes presented in Section 13.4.7 provides a more general and elegant way to control placement in a specific PR.

13.2.3 Spill Code

When the allocator decides that it cannot keep some LR in a register, it must spill that LR to memory before reallocating its PR to another value. It must also restore the spilled LR into a PR before any subsequent use. These added spills and restores increase execution time, so the allocator should insert as few of them as practical. The most direct measure of allocation quality is the time spent in spill code at runtime.

Allocators differ in the granularity with which they spill values. The global allocator described in Section 13.4 spills the entire live range. When it decides to spill LR_i, it inserts a spill after each definition in LR_i and a restore immediately before each use of LR_i. In effect, it breaks LR_i into a set of tiny LRs, one at each definition and each use.

By contrast, the local allocator described in Section 13.3 spills a live range only between the point where its PR is reallocated and its next use. Because it operates in a single block, with straight-line control flow, it can easily implement this policy; the LR has a unique next use and the point of spill always precedes that use.

Between these two policies, "spill everywhere" and "spill once," lie many possible alternatives. Researchers have experimented with spilling partial live ranges. The problem of selecting an optimal granularity for spilling is, undoubtedly, as hard as finding an optimal allocation; the correct granularity likely differs between live ranges. Section 13.5 describes some of the schemes that compiler writers and researchers have tried.

Nonuniform Spill Costs

An LR might be clean due to a prior spill along the current path, or because its value also exists in memory.

To further complicate spilling, the allocator should account for the fact that properties of an LR can change the cost to spill it and to restore it.

Dirty Value In the general case, the LR contains a value that has been computed and has not yet been stored to memory; we say that the LR is *dirty*. A dirty LR requires a store at its spill points and a load at its restore points.

Clean Value If the LR's value already exists in memory, then a spill does not require a store; we say that the LR is *clean*. A clean LR costs nothing to spill; its restores cost the same as those of a dirty value.

Rematerializable Value Some LRs contain values that cost less to recompute than to spill and restore. If the values used to compute the LR's value are available at each use of the LR, the allocator can simply recompute the LR's value on demand. A classic example is an LR defined by an immediate load. Such an LR costs nothing to spill; to restore it, the compiler inserts the recomputation. Typically, an immediate load is cheaper than a load from memory.

The allocator should, to the extent possible, account for the nonuniform nature of spill costs. Of course, doing so complicates the allocator. Furthermore, the NP-complete nature of allocation suggests that no simple heuristic will make the best choice in every situation.

Spill Locations

When the allocator spills a dirty value, it must place the value somewhere in memory. If the LR corresponds precisely to a variable kept in memory, the allocator could spill the value back to that location. Otherwise, it must reserve a location in memory to hold the value during the time when the value is live and not in a PR.

Spill location
a memory address associated with an LR that holds its value when the LR has no PR

Most allocators place spill locations at the end of the procedure's local data area. This strategy allows the spill and restore code to access the value at a fixed offset from the ARP, using an address-immediate memory operation if the ISA provides one. The allocator simply increases the size of the local data area, at compile time, so the allocation incurs no direct runtime cost.

Note that any value in a spill location is unambiguous, an important point for postal-location scheduling.

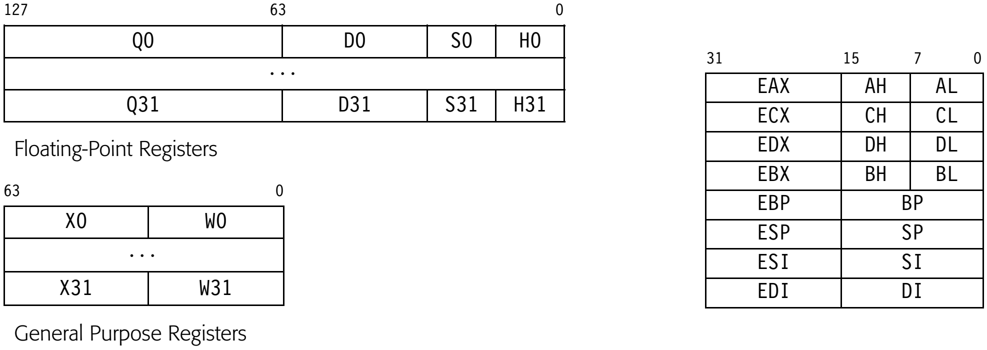
Because an LR is only of interest during that portion of the code where it is live, the allocator has the opportunity to reduce the amount of spill memory that it uses. If LR_i and LR_j do not interfere, they can share the same spill location. Thus, the allocator can also use the interference graph to color spill locations and reduce memory use for spills.

13.2.4 Register Classes

Many processors support multiple classes of registers. For example, most ISAs have a set of general purpose registers (GPRs) for use in integer operations and address manipulation, and another set of floating-point registers (FPRs). In the case of GPRs and FPRs, the two register classes are, almost always, implemented with physically and logically disjoint register sets.

Register class
a distinct group of named registers that share common properties, such as length and supported operations

Often, an ISA will overlay multiple register classes onto a single physical register set. As shown in Fig. 13.2(a), the ARM A-64 supports four sizes of floating-point values in one set of quad-precision (128 bit) FPRs. The 128-bit FPRs are named `Q0`, `Q1`, ..., `Q31`. Each `Qi` is overlaid with a 64-bit register `Di`, a 32-bit register `Si`, and a 16-bit register `Hi`. The shorter registers occupy the low-order bits of the longer registers.



(a) ARM-A64 Register Names

(b) Intel IA-32 Register Names

■ **FIGURE 13.2** Overlapping Register Names.

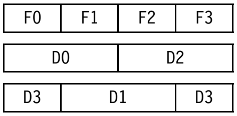
The ARM A-64 GPRs follow a similar scheme. The 64-bit GPRs have both 64-bit names Xi and 32-bit names Wi. Again, the 32-bit field occupies the low order bits of the 64-bit register.

The discussion focuses on a subset of the IA-32 register set. It ignores segment registers and most of the registers added in IA-64.

The Intel IA-32 has a small register set, part of which is depicted in Fig. 13.2(b). It provides eight 32-bit registers. The CISC instruction set uses distinct registers for specific purposes, leading to unique names for each register, as shown. For backward compatibility with earlier 16-bit processors, the PR set supports naming subfields of the 32-bit registers.

- In four of the registers, the programmer can name the 32-bit register, its lower 16 bits, and two 8-bit fields. These registers are the accumulator (EAX), the count register (ECX), the data register (EDX), and the base register (EBX).
- In the other four registers, the programmer can name both the 32-bit register and its lower 16 bits. These registers are the base of stack (EBP), the stack pointer (ESP), the string source index pointer (ESI), and the string destination index pointer (EDI).

The figure omits the instruction pointer (EIP and IP) and flag register (EFLAGS and FLAGS), which have both 32-bit and 16-names. The later IA-64 features a larger set of 32-bit GPRs, but preserves the IA-32 names and features in the low numbered registers.



Register Pairs

D3 is shown as wrapping around the end of the register set.

Many earlier ISAs used pairing schemes in the FPR set. The drawing in the margin shows how a four register set might work. It would consist of the four 32-bit PRs, F0, F1, F2, and F3. 64-bit values occupy a pair of adjacent registers. If a register pair can begin with any register, then four pairs are possible: D0, D1, D2, and D3.

Some ISAs restrict a register-pair to begin with an odd-numbered register—an aligned pair. With aligned pairs, only the registers shown as `D0` and `D2` would be available. With aligned pairs, use of `D0` precludes the use of `F0` and `F1`. With unaligned pairs, use of `D0` still precludes the use of `F0` and `F1`. It also precludes the use of `D1` and `D3`.

In general, the register allocator should make effective use of all available registers. Thus, it must understand the processor's register classes and include mechanisms to use them in a fair and efficient manner. For physically disjoint classes, such as floating-point and general purpose register classes, the allocator can simply allocate them independently. If floating-point spills use GPRs for address calculations, the compiler should allocate the FPRs first.

The design of the register-set name space affects the difficulty of managing register classes in the allocator. For example, the ARM A-64 naming scheme allows the allocator to treat all of the fields in a single PR as a single resource; it can use one of `X0` or `W0`. By contrast, the IA-32 allows concurrent use of both `AH` and `AL`. Thus, the allocator needs more complex mechanisms to handle the IA-32 register set. Section 13.4.7 explores how to build such mechanisms into a global graph-coloring register allocator.

SECTION REVIEW

The register allocator must decide, at each point in the code, which values should be kept in registers. To do so, it computes a name space for the values in the code, often called live ranges. The allocator must discover which live ranges cannot share a register—that is, which live ranges interfere with each other. Finally, it must assign some live ranges to registers and relegate some to memory. It must insert appropriate loads and stores to move values between registers and memory to enforce its decisions.

REVIEW QUESTIONS

1. Consider a block of straight-line code where the largest register pressure at an operation in the block is j. Assume that the allocator is allowed to use k registers. If $j = k$, can the allocator map the live ranges onto the PRs without spilling?
2. Consider a procedure represented as n ILOC operations. Can you bound the number of nodes and edges in the interference graph?

13.3 LOCAL REGISTER ALLOCATION

Recall that a *basic block* is a maximal length sequence of straight-line code.

The simplest formulation of the register allocation problem is local allocation: consider a single basic block and a single class of *k* PRs. This problem captures many of the complexities of allocation and serves as a useful introduction to the concepts and terminology needed to discuss global allocation. To simplify the discussion, we will assume that one block constitutes the entire program.

The input code uses *source registers*, written in code as $\texttt{sr}_i$.

The output code uses *physical registers*, written in code as either $\texttt{pr}_i$ or simply $\texttt{r}_i$.

The physical registers correspond, in general, to named registers in the target ISA.

The input block contains a series of three-address operations, each of which has the form $\texttt{op}_i\ \texttt{sr}_i, \texttt{sr}_j \Rightarrow \texttt{sr}_m$. From a high-level view, the local register allocator rewrites the block to replace each reference to a source register (SR) with a reference to a specific physical register (PR). The allocator must preserve the input block's original meaning while it fits the computation into the *k* PRs provided by the target machine.

If, at any point in the block, the computation has more than *k* live values—that is, values that may be used in the future—then some of those values will need to reside in memory for some portion of their lifetimes. (*k* registers can hold at most *k* values.) Thus, the allocator must insert code into the block to move values between memory and registers as needed to ensure that all values are in PRs when needed and that no point in the code needs more than *k* PRs.

This section presents a version of Best's algorithm, which dates back to the original FORTRAN compiler. It is one of the strongest known local allocation algorithms. It makes two passes over the code. The first pass derives detailed knowledge about the definitions and uses of values; essentially, it computes LIVE information within the block. The second pass then performs the actual allocation.

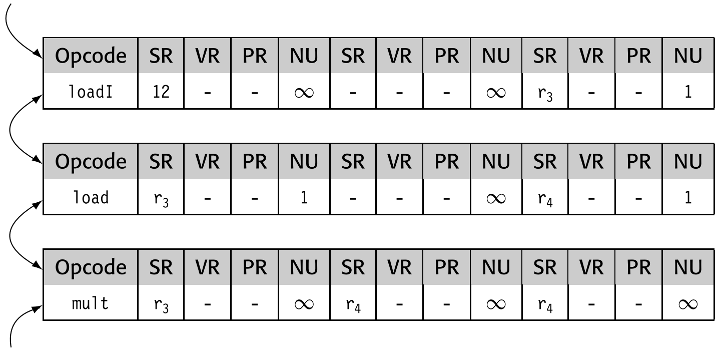
Spill
When the allocator moves a live value from a PR to memory, it *spills* the value.

Restore
When the allocator retrieves a previously spilled value from memory, it *restores* the value.

Best's algorithm has one guiding principle: when the allocator needs a PR and they are all occupied, it should spill the PR that contains the value whose next use is farthest in the future. The intuition is clear; the algorithm chooses the PR that will reduce demand for PRs over the longest interval. If all values have the same cost to spill and restore, this choice is optimal. In practice, that assumption is rarely true, but Best's algorithm still does quite well.

To explain the algorithm it helps to have a concrete data structure. Assume a three-address, ILOC-like code, represented as a list of operations. Each operation, such as $\texttt{mult}\ \texttt{sr}_1, \texttt{sr}_2 \Rightarrow \texttt{sr}_3$ is represented with a structure:

	OPERAND 1				OPERAND 2				OPERAND 3			
Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
mult	r_1	-	-	∞	r_2	-	-	∞	r_3	-	-	∞



Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
`loadI`	12	-	-	∞	-	-	-	∞	r_3	-	-	1

Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
`load`	r_3	-	-	1	-	-	-	∞	r_4	-	-	1

Opcode	SR	VR	PR	NU	SR	VR	PR	NU	SR	VR	PR	NU
`mult`	r_3	-	-	∞	r_4	-	-	∞	r_4	-	-	∞

■ **FIGURE 13.3** Representing a List of Operations.

The operation has an opcode, two inputs (operands 1 and 2), and a result (operand 3). Each operand has a source-register name (SR), a virtual-register name (VR), a physical-register name (PR), and the index of its next use (NU).

Register allocation is, at its core, the process of constructing a new name space and modifying the code to use that space. Keeping the SR, VR, and PR names separate simplifies both writing and debugging the allocator.

A list of operations might be represented as a doubly linked list, as shown in Fig. 13.3. The local allocator will need to traverse the list in both directions. The list could be created in an array of structure elements, or with individually allocated or block-allocated structures.

The first operation, a `loadI`, has an immediate value as its first argument, stored in the SR field. It has no second argument. The next operation, a `load`, also has just one argument. The final operation, a `mult`, has two arguments. Because the code fragment does not contain a next use for any of the registers mentioned in the `mult` operation, their NU fields are set to ∞.

Since the meaning is clear, we store a `loadI`'s constant in its first operand's SR field.

13.3.1 Renaming in the Local Allocator

To simplify the local allocator's implementation, the compiler can first rename SRs so that they correspond to live ranges. In a single block, an LR consists of a single definition and one or more uses. The *span* of the LR is the interval in the block between its definition and its last use.

The renaming algorithm finds the live range of each value in a block. It assigns each LR a new name, its VR name. Finally, it rewrites the code in terms of VRs. Renaming creates a one-to-one correspondence between LRs and VRs which, in turn, simplifies many of the data structures in the local allocator. The allocator then reasons about VRs, rather than arbitrary SR names.

```
VRName ← 0
for i ← 0 to max source-register number do
    SRToVR[i] ← invalid
    PrevUse[i] ← ∞

index ← block length
for each Op in the block, bottom to top, do
    for each operand, O, that OP defines do      // defs first
        if SRToVR[O.SR] = invalid then           // def has no uses
            SRToVR[O.SR] ← VRName++              // start a new VR anyway
        O.VR ← SRToVR[O.SR]                      // set VR and NU for O
        O.NU ← PrevUse[O.SR]
        PrevUse[O.SR] ← ∞
        SRToVR[O.SR] ← invalid                   // next use of SR starts new VR
    for each operand, O, that OP uses do         // uses after defs
        if SRToVR[O.SR] = invalid then           // start a new VR
            SRToVR[O.SR] ← VRName++
        O.VR ← SRToVR[O.SR]                      // set VR and NU for O
        O.NU ← PrevUse[O.SR]
    for each operand, O, that OP uses do
        PrevUse[O.SR] ← index                    // save to set next NU
    index ← index - 1
```

■ **FIGURE 13.4** Renaming Source Registers into Live Ranges.

The compiler can discover live ranges and rename them into VRs in a single backward pass over the block. As it does so, it can also collect and record next use information for each definition and use in the block. The algorithm, shown in Fig. 13.4, assumes the representation described in the previous section.

The renaming algorithm builds two maps: *SRToVR*, which maps an SR name to a VR name, and *PrevUse*, which maps an SR name into the ordinal number of its most recent use. The algorithm begins by initializing each *SRToVR* entry to *invalid* and each *PrevUse* entry to ∞.

The algorithm walks the block from the last operation to the first operation. At each operation, it visits definitions first and then uses. At each operand, it updates the maps and defines the *VR* and *NU* fields.

If the SR for a definition has no VR, that value is never used. The algorithm still assigns a VR to the SR.

When the algorithm visits a use or def, it first checks whether or not the reference's SR, *O.SR*, already has a VR. If not, it assigns the next available VR name to the SR and records that fact in *SRToVR[O.SR]*. Next, it records the VR name and next use information in the operand's record. If the operand is a use, it sets *PrevUse[O.SR]* to the current operation's index. For a definition, it sets *PrevUse[O.SR]* back to ∞.

The algorithm visits definitions before uses to ensure that the maps are updated correctly in cases where an SR name appears as both a definition and a use. For example, in `add` r_{17}, $r_{18} \Rightarrow r_{18}$, the algorithm will rewrite the definition with *SRToVR[*r_{18}*]*; update *SRToVR[*r_{18}*]* with a new VR name for the use; and then set *PrevUse[*r_{18}*]* to ∞.

Note that all operands to a `store` are uses. The `store` defines a memory location, not a register.

After renaming, each live range has a unique VR name. An SR name that is defined in multiple places is rewritten as multiple distinct VRs. In addition, each operand in the block has its *NU* field set to either the ordinal number of the next operation in the block that uses its value, or ∞ if it has no next use. The allocator uses this information when it chooses which VRs to spill.

After renaming, we use *live range* and *virtual register* interchangeably.

Consider, again, the code from Fig. 13.1. Panel (a) shows the original code. Panel (b) shows that code after renaming. Panel (c) shows the span of each live range, as an interval graph. The allocator does not rename r_{arp} because it is a dedicated PR that holds the activation record pointer and, thus, not under the allocator's control.

The maximum demand for registers, MAXLIVE, occurs at the start of the first `mult` operation, marked in panel (c) by the dashed gray line. Six VRs are live at that point. Both VR_7 and VR_8 are live at the start of the operation. The operation is the last use of VR_7 and VR_8, as well as the definition of VR_5.

MAXLIVE
The maximum number of concurrently live VRs in a block

13.3.2 Allocation and Assignment

The algorithm for the local allocator appears in Fig. 13.5. It performs allocation and assignment in a single forward pass over the block. It starts with an assumption that no values are in PRs. It iterates through the operations, in order, and incrementally allocates a PR to each VR. At each operation, the allocator performs three steps.

1. To ensure that a use has a PR, the allocator first looks for a PR in the *VRToPR* map. If the entry for VR is *invalid*, the algorithm calls *GetAPR* to find a PR. The allocator uses a simple marking scheme to avoid allocating the same PR to conflicting uses in a single operation.
2. In the second step, the allocator determines if any of the uses are the last use of the VR. If so, it can free the PR, which makes the PR available for reassignment, either to a result of the current operation or to some VR in a future operation.
3. In the third step, the allocator ensures that each VR defined by the operation has a PR allocated to hold its value. Again, the allocator uses *GetAPR* to find a suitable register.

If a single instruction contains multiple operations, the allocator should process all of the uses before any of the definitions.

```
for vr ← 0 to max VR number do
    VRToPR[vr] ← invalid

for pr ← 0 to max PR number do
    PRToVR[pr] ← invalid
    PRNU[pr] ← ∞
    push(pr)    // pop() occurs in GetAPR()

// iterate over the block
for each OP in the block, in linear order, do
    clear the mark in each PR              // reset marks

    for each use, U, in OP do              // allocate uses
        pr ← VRToPR[U.VR]
        if (pr = invalid) then
            U.PR ← GetAPR(U.VR,U.NU)
            Restore(U.VR,U.PR)
        else
            U.PR ← pr
        set the mark in U.PR

    for each use, U, in OP do              // last use?
        if (U.NU = ∞ and PRToVR[U.PR] ≠ invalid) then
            FreeAPR(U.PR)

    clear the mark in each PR              // reset marks

    for each definition, D, in OP do       // allocate defs
        D.PR ← GetAPR(D.VR, D.NU)
        set the mark in D.PR
```

```
GetAPR(vr, nu)
    if stack is nonempty then
        x ← pop()
    else
        pick an unmarked x to spill
        Spill(x)

    VRToPR[vr] ← x
    PRToVR[x] ← vr
    PRNU[x] ← nu

    return x


FreeAPR(pr)
    VRToPR[PRToVR[pr]] ← invalid
    PRtoVR[pr] ← invalid
    PRNU[pr] ← ∞
    push(pr)
```

■ **FIGURE 13.5** The Local Allocator.

Each of these steps is straightforward, except for picking the value to spill. Most of the complexity of local allocation falls in that task.

The Workings of *GetAPR*

As it processes an operation, the allocator will need to find a PR for any VR v that does not currently have one. This act is the essential act of register allocation. Two situations arise:

1. *Some PR p is free*: The allocator can assign p to v. The algorithm maintains a stack of free PRs for efficiency.
2. *No PR is free*: The allocator must choose a VR to evict from its PR p, save the value in p to its spill location, and assign p to hold v.

If the reference to v is a use, the allocator must then restore v's value from its memory location to p.

SPILL AND RESTORE CODE

At the point where the allocator inserts spill code, all of the physical registers (PRs) are in use. The compiler writer must ensure that the allocator can still spill a value.

Two scenarios are possible. Most likely, the target machine's ISA supports an address mode that allows the spill without need for an additional PR. For example, if the ARP has a dedicated register, say r_{arp}, and the ISA includes an address-immediate store operation, like ILOC's `storeAI`, then spill locations in the local data area can be reached without an additional PR.

On a target machine that only supports a simple load and store, or an implementation where spill locations cannot reside in the activation record, the compiler would need to reserve a PR for the address computation, reducing the pool of available PRs. Of course, the reserved register is only needed if MAXLIVE $> k$. (If MAXLIVE $\leq k$, then no spills are needed and neither is the reserved register.)

Best's heuristic states that the allocator should spill the PR whose current VR has the farthest next use. The algorithm maintains *PRNU* to facilitate this decision. It simply chooses the PR with the largest *PRNU*. If the allocator finds two PRs with the same *PRNU*, it must choose one.

The implementation of *PRNU* is a tradeoff between the efficiency of updates and the efficiency of searches. The algorithm updates *PRNU* at each register reference. It searches *PRNU* at each spill. As shown, *PRNU* is a simple array; if updates are much more frequent than spills, that makes sense. If spills are frequent enough, a priority queue for *PRNU* may improve allocation time.

Tracking Physical and Virtual Registers

To track the relationship between VRs and PRs, the allocator maintains two maps. *VRToPR* contains, for each VR, either the name of the PR to which it is currently assigned, or the value *invalid*. *PRToVR* contains, for each PR, either the name of the VR to which it is currently assigned, or the value *invalid*.

As it proceeds through the block, the allocator updates these two maps so that the following invariant always holds:

if $\mathit{VRToPR}[vr] \neq \mathit{invalid}$ then $\mathit{PRToVR}[\mathit{VRToPR}[vr]] = vr$.

The code in *GetAPR* and *FreeAPR* maintains these maps to ensure that the invariant holds true. In addition, these two routines maintain *PRNU*, which maps a PR into the ordinal number of the operation where it is next used—a proxy for distance to that next use.

			`restore` x_3
`spill` x_2		`spill` x_2	`restore` x_1
`restore` x_3	`restore` x_3	`restore` x_3	`restore` x_3
`restore` x_2	`restore` x_1	`restore` x_2	`restore` x_1
Spill Dirty	Spill Clean	Spill Dirty	Spill Clean

(a) References x_3 x_1 x_2

(b) References x_3 x_1 x_3 x_1 x_2

■ **FIGURE 13.6** Spills of Clean Versus Dirty Values.

Spills and Restores

Conceptually, the implementation of *Spill* and *Restore* from Fig. 13.5 can be quite simple.

Spill locations typically are placed at the end of the local data area in the activation record.

- To spill a PR p, the allocator can use *PRToVR* to find the VR v that currently lives in p. If v does not yet have a spill location, the allocator assigns it one. Next, it emits an operation to store p into the spill location. Finally, it updates the maps: *VRToPR*, *PRToVR*, and *PRNU*.
- To restore a VR v into a PR p, the allocator simply generates a load from v's spill location into p. As the final step, it updates the maps: *VRToPR*, *PRToVR*, and *PRNU*.

If all spills have the same cost and all restores have the same cost, then Best's algorithm generates an optimal allocation for a block.

Complications from Spill Costs

In real programs, spill and restore costs are not uniform. Real code contains both clean and dirty values; the cost to spill a dirty value is greater than the cost to spill a clean value. To see this, consider running the local allocator on a block that contains references to three names, x_1, x_2, and x_3, with just two PRs ($k = 2$).

Assume that the register allocator is at a point where x_1 and x_2 are currently in registers and x_1 is clean and x_2 is dirty. Fig. 13.6 shows how different spill choices affect the code in two different scenarios.

Reference string
A reference string is just a list of references to registers or addresses. In this context, each reference is a use, not a definition.

Panel (a) considers the case when the reference string for the rest of the block is x_3 x_1 x_2. If the allocator consistently spills clean values before dirty values, it introduces less spill code for this reference string.

Panel (b) considers the case when the reference string for the rest of the block is x_3 x_1 x_3 x_1 x_2. Here, if the allocator consistently spills clean values before dirty values, it introduces more spill code.

The presence of both clean and dirty values fundamentally changes the local allocation problem. Once the allocator faces two kinds of values with different spill costs, the problem becomes NP-hard. The introduction of rematerializable values, which makes restore costs nonuniform, makes the problem even more complex. Thus, a fast deterministic allocator cannot always make optimal spill decisions. However, these local allocators can produce good allocations by choosing among LRs with different spill costs with relatively simple heuristics.

In practice, the allocator may produce better allocations if it differentiates between dirty, clean, and rematerializable values (see Section 13.2.3). If two PRs have the same distance to their next uses and different spill costs, then the allocator should spill the lower-cost PR.

Remember, however, that the problem is NP-hard. No efficient, deterministic algorithm will always produce optimal results.

The issue becomes more complex, however, in choosing between PRs with different spill costs that have next-use distances that are close but not identical. For example, given a dirty value with next use of n and a rematerializable value with next use of $n - 1$, the latter value will sometimes be the better choice.

SECTION REVIEW

The limited context in local register allocation simplifies the problem enough so that a fast, intuitive algorithm can produce high-quality allocations. The local allocator described in this section operates on a simple principle: *when a PR is needed, spill the PR whose next use is farthest in the future*.

In a block where all values had the same spill costs, the local allocator would achieve optimal results. When the allocator must contend with both dirty and clean values, the problem becomes combinatorially hard. A local allocator can produce good results, but it cannot guarantee optimal results.

REVIEW QUESTIONS

1. Modify the renaming algorithm, shown in Fig. 13.4, so that is also computes MAXLIVE, the maximum number of simultaneously live values at any instruction in the block.
2. Rematerializing a known constant is an easy decision, because the spill requires no code and the restore is a single load immediate operation. Under what conditions could the allocator profitably rematerialize an operation such as `add` $r_a, r_b \Rightarrow r_x$?

13.4 GLOBAL ALLOCATION VIA COLORING

Most compilers use a global register allocator—one that considers the entire procedure as context. The global allocator must account for more complex control flow than does a local allocator. Live ranges have multiple definitions and uses; they form complex webs across the control-flow graph. Different blocks execute different numbers of times, which complicates spill cost estimation. While some of the intuitions from local allocation carry forward, the algorithms for global allocation are much more involved.

A global allocator faces another complication: it must coordinate register use across blocks. In the local algorithm, the mapping from an enregistered LR to a PR is, essentially, arbitrary. By contrast, a global allocator must either keep an LR in the same register across all of the blocks where it is live or insert code to move it between registers.

Spilling an LR breaks it into small pieces that can be kept in distinct PRs.

Most global allocation schemes build on a single paradigm. They represent conflicts between register uses with an interference graph and then color that graph to find an allocation. Within this model, the compiler writer faces a number of choices. Live ranges may be shorter or longer. The graph may be precise or approximate. When the allocator must spill, it can spill that LR everywhere or it can spill the LR only in regions of high register pressure.

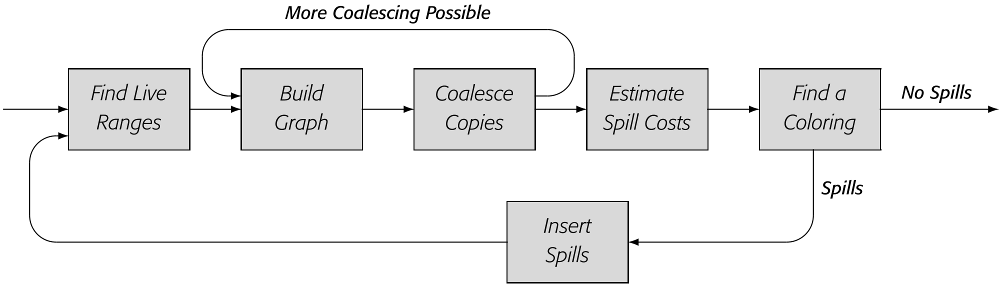
These choices create a variety of different specific algorithms. This section focuses on one specific set of choices: maximal length live ranges, a precise interference graph, and a spill-everywhere discipline. These choices define the global coloring allocator. Section 13.5 explores variations and improvements to the global coloring allocator.

Fig. 13.7 shows the structure of the global coloring allocator.

Find Live Ranges The allocator finds live ranges and rewrites the code with a unique name for each LR. The new name space ensures that distinct values are not constrained simply because they shared the same name in the input code.

Build Graph The allocator builds the interference graph. It creates a node for each LR and adds edges from LR_i to any LR_j that is live at an operation that defines LR_i, unless the operation is a copy. Building the graph tends to dominate the cost of allocation.

Coalesce Copies The allocator looks at each copy operation, $LR_i \Rightarrow LR_j$. If LR_i and LR_j do not interfere, it combines the LRs, removes the copy, and updates the graph. Coalescing reduces the number of LRs and reduces the degree of other nodes in the graph.



■ **FIGURE 13.7** Structure of the Global Coloring Allocator.

Unfortunately, the graph update is conservative rather than precise (see Section 13.4.3). Thus, if any LRs are combined, the allocator iterates the *Build–Coalesce* process until it cannot coalesce any more LRs—typically two to four iterations.

Estimate Spill Costs The allocator computes, for each LR, an estimate of the runtime cost of spilling the entire LR. It adds the costs of the spills and restores, each multiplied by the estimated execution frequency of the block where the code would be inserted.

The spill cost computation has many corner cases (see Sections 13.2.3 and 13.4.4).

Find a Coloring The allocator tries to construct a *k*-coloring for the interference graph. It uses a two-phase process: graph simplification to construct an order for coloring, then graph reconstruction that assigns colors as it reinserts each node back into the graph.

If the allocator finds a *k*-coloring, it rewrites the code and exits. If any nodes remain uncolored, the allocator invokes *Insert Spills* to spill the uncolored LRs. It then restarts the allocator on the modified code.

The second and subsequent attempts at coloring take less time than the first try because coalescing in the first pass has reduced the size of both the problem and the interference graph.

Insert Spills For each uncolored LR, the allocator inserts a spill after each definition and a restore before each use. This converts the uncolored LR into a set of tiny LRs, one at each reference. This modified program is easier to color than the original code.

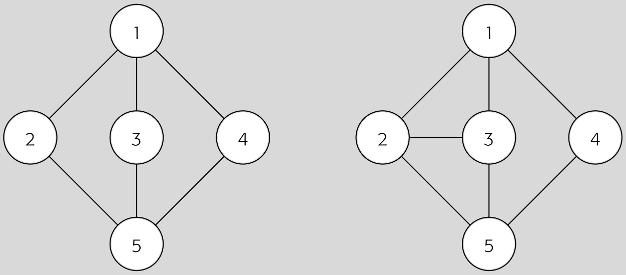
The following subsections describe these phases in more detail.

13.4.1 Find Global Live Ranges

As its first step, the global allocator constructs maximal-sized global live ranges (see Section 13.2.1). A global LR is a set of definitions and uses that contains all of the uses that a definition in the set can reach, along with all

GRAPH COLORING

Graph coloring is a common paradigm for global register allocation. For an arbitrary graph *G*, a coloring of *G* assigns a color to each node in *G* so that no pair of adjacent nodes has the same color. A coloring that uses *k* colors is termed a *k*-coloring, and the smallest such *k* for a given graph is called the graph's *minimum chromatic number*. Consider the following graphs:



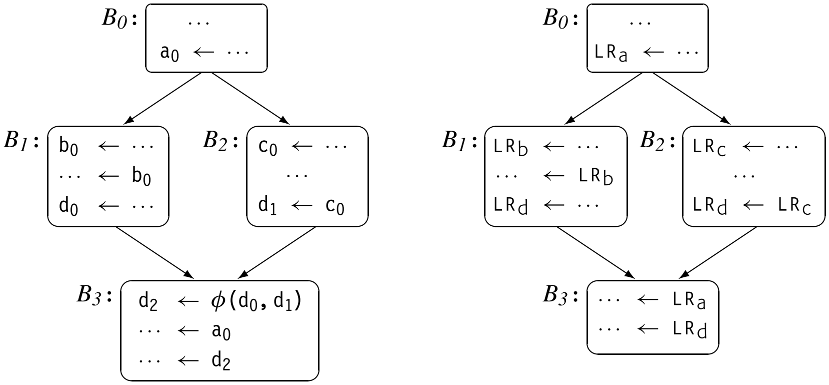
The graph on the left is two-colorable. For example, we can assign *blue* to nodes 1 and 5, and *red* to nodes 2, 3, and 4. Adding the edge (2, 3), as shown on the right, makes the graph three-colorable, but not two-colorable. (Assign *blue* to nodes 1 and 5, *red* to nodes 2 and 4, and *yellow* to node 3.)

For a given graph, finding its minimum chromatic number is NP-complete. Similarly, determining if a graph is *k*-colorable, for fixed *k*, is NP-complete. Graph coloring allocators use approximate methods to find colorings that fit the available resources.

The maximum degree of any node in a graph gives an upper bound on the graph's chromatic number. A graph with maximum degree of x can always be colored with $x + 1$ colors. The two graphs shown above demonstrate that degree is a loose upper bound. Both graphs have maximum degree of three. Both graphs have colorings with fewer than four colors. In each case, high-degree nodes have neighbors that can receive the same color.

of the definitions that can reach those uses. Thus, the LR forms a complex web of definitions and uses that, ideally, should reside in a single PR.

The algorithm to construct live ranges is straightforward if the allocator can work from the SSA form of the code. Thus, the first step in finding live ranges is to convert the input code to SSA form, if necessary. The allocator can then build maximal-sized live ranges with a simple approach: at each ϕ-function, combine all of the names, both definition and uses, into a single LR. If the allocator applies this rule at each ϕ-function, it creates the set of maximal global LRs.



(a) Code Fragment in Pruned SSA Form

(b) Rewritten in Terms of Live Ranges

■ **FIGURE 13.8** Discovering Live Ranges.

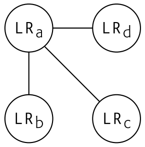
To make this process efficient, the compiler can use the disjoint-set union-find algorithm. To start, it assigns a unique set name to each SSA name. Next, it visits each ϕ-function in the code and unions together the sets associated with each ϕ-function parameter and the set for the ϕ-function result. After all of the ϕ-functions have been processed, each remaining unique set becomes an LR. The allocator can either rewrite the code to use LR names or it can create and maintain a mapping between SSA names and LR names. In practice, the former approach seems simpler.

Since the process of finding LRs does not move any definitions or uses, translation out of SSA form is trivial. The LR name space captures the effects that would require new copies during out-of-SSA translation. Thus, the compiler can simply drop the ϕ-functions and SSA-names during renaming.

Fig. 13.8(a) shows a code fragment in semipruned SSA form that involves source-code variables, a, b, c, and d. To find the live ranges, the allocator assigns each SSA name a set that contains its name. It unions together the sets associated with names used in the ϕ-function, $\{d_0\} \cup \{d_1\} \cup \{d_2\}$. This gives a final set of four LRs: $LR_a = \{a_0\}$, $LR_b = \{b_0\}$, $LR_c = \{c_0\}$, and $LR_d = \{d_0, d_1, d_2\}$. Fig. 13.8(b) shows the code rewritten to use the LRs.

13.4.2 **Build an Interference Graph**

To model interferences, the global allocator builds an interference graph, $I = (N, E)$. Nodes in N represent individual live ranges and edges in E represent interferences between live ranges. Thus, an undirected edge $(n_i, n_j) \in E$ exists if and only if the corresponding live ranges LR_i and LR_j interfere. The interference graph for the code in Fig. 13.8(b) appears in the margin.



An Interference Graph

```
for each $LR_i$ do
    create a node $n_i \in N$

for each basic block b do
    LiveNow ← LiveOut(b)

    for each operation i in b, from bottom to top,
        assuming form $op_i$  $LR_a$, $LR_b \Rightarrow LR_c$  do
        remove $LR_c$ from LiveNow
        for each $LR_i \in$ LiveNow† do
            add ($LR_i$, $LR_c$) to E
        add $LR_a$ and $LR_b$ to LiveNow
```

† If the operation is a copy, $LR_i \Rightarrow LR_j$, do not add the edge (LR_i,LR_j).

■ **FIGURE 13.9** Constructing the Interference Graph.

LR_a interferes with each of LR_b, LR_c, and LR_d. None of LR_b, LR_c, or LR_d interfere with each other; they could share a single PR.

If the compiler can color *I* with *k* or fewer colors, then it can map the colors directly onto PRs to produce a legal allocation. In the example, LR_a interferes with each of LR_b, LR_c, and LR_d. In a coloring, LR_a must receive its own color and, in an allocation, it cannot share a PR with LR_b, LR_c, or LR_d. The other live ranges do not interfere with each other. Thus, LR_b, LR_c, and LR_d could share a single color and, in the code, a single PR. This interference graph is two-colorable, and the code can be rewritten to use just two registers.

```
B1 :  LRb ← ···
      LRd ← ···
      ··· ← LRb
```

B_1 with the Definitions Reordered

Now, consider what would happen if another phase of the compiler reordered the last two definitions in B_1, as shown in the margin. This change makes LR_b live at the definition of LR_d. It adds an edge (LR_b,LR_d) to the interference graph, which makes the graph three-colorable rather than two-colorable. (The graph is small enough to prove this by enumeration.) With this new graph, the allocator has two options: to use three registers, or, if only two registers are available, to spill one of LR_b or LR_a before the definition of LR_d in B_1. Alternatively, the allocator could reorder the two operations and eliminate the interference between LR_b and LR_d. Typically, register allocators do not reorder operations. Instead, allocators assume a fixed order of operations and leave ordering questions to the instruction scheduler (see Chapter 12).

Section 11.3.2 also uses LiveNow.

Given the code, renamed into LRs, and LiveOut sets for each block in the renamed code, the allocator can build the interference graph in one pass over each block, as shown in Fig. 13.9. The algorithm walks the block, from bottom to top. At each operation, it computes LiveNow, the set of values that are live at the current operation. At the bottom of the block, LiveOut and LiveNow must be identical. As the algorithm walks backward through the block, it adds the appropriate interference edges to the graph and updates the LIVENOW set to reflect each operation's impact.

The algorithm implements the definition of interference given earlier: LR_i and LR_j interfere only if LR_i is live at a definition of LR_j, or vice versa. The allocator adds, at each operation, an interference between the defined LR and each LR that is live after the operation.

Copy operations require special treatment. A copy $LR_i \Rightarrow LR_j$ does not create an interference between LR_i and LR_j because the two live ranges have the same value after the copy executes and, thus, could occupy the same register. If subsequent context creates an interference between these live ranges, that operation will create the edge. Treating copies in this way creates an interference graph that precisely captures when LR_i and LR_j can occupy the same register. As the allocator encounters copies, it should create a list of all the copy operations for later use in coalescing.

To improve the allocator's efficiency, it should build both a lower-triangular bit matrix and a set of adjacency lists to represent *E*. The bit matrix allows a constant-time test for interference, while the adjacency lists allow efficient iteration over a node's neighbors. The two-representation strategy uses more space than a single representation would, but pays off in reduced allocation time. As suggested in Section 13.2.4, the allocator can build separate graphs for disjoint register classes, which reduces the maximum graph size.

Insertion into the lists should check the bit-matrix to avoid duplication.

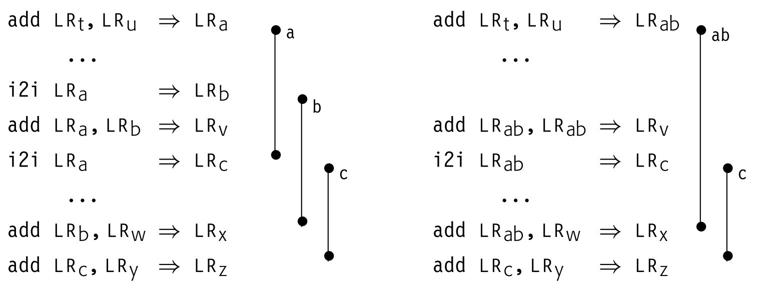
13.4.3 Coalesce Copy Operations

The allocator can use the interference graph to implement a strong form of copy coalescing. If the code contains a copy operation, $LR_i \Rightarrow LR_j$, and the allocator can determine that LR_i and LR_j do not interfere at some other operation, then the allocator can combine the LRs and remove the copy operation. We say that the copy has been "coalesced."

Coalescing a copy has several beneficial effects. It eliminates the actual copy operation, which makes the code smaller and, potentially, faster. It reduces the degree of any LR that previously interfered with both LR_i and LR_j. It removes a node from the graph. Each of these effects makes the coloring pass faster and more effective.

In his thesis, Briggs shows examples where coalescing eliminates up to one-third of the initial live ranges.

Fig. 13.10 shows a simple, single-block example. The original code appears in panel (a). Intervals to the right indicate the extents of the live ranges that are involved in the copy operation. Even though LR_a overlaps both LR_b and LR_c, it interferes with neither of them because the overlaps involve copy operations. Since LR_b is live at the definition of LR_c, LR_b and LR_c interfere. Both copy operations are candidates for coalescing.



add $LR_t, LR_u \Rightarrow LR_a$
...
i2i $LR_a \Rightarrow LR_b$
add $LR_a, LR_b \Rightarrow LR_v$
i2i $LR_a \Rightarrow LR_c$
...
add $LR_b, LR_w \Rightarrow LR_x$
add $LR_c, LR_y \Rightarrow LR_z$

(a) Before Coalescing

add $LR_t, LR_u \Rightarrow LR_{ab}$
...
add $LR_{ab}, LR_{ab} \Rightarrow LR_v$
i2i $LR_{ab} \Rightarrow LR_c$
...
add $LR_{ab}, LR_w \Rightarrow LR_x$
add $LR_c, LR_y \Rightarrow LR_z$

(b) After Coalescing LR_a and LR_b

■ **FIGURE 13.10** Combining Live Ranges by Coalescing Copy Operations.

Fig. 13.10(b) shows the result of coalescing LR_a and LR_b to produce LR_{ab}. LR_{ab} and LR_c still do not interfere, because LR_c is created by the copy operation from LR_{ab}. Combining LR_a and LR_b reduces LR°_{v} by one. Before coalescing, both LR_a and LR_b interfered with LR_v. After coalescing, those values occupy a single LR rather than two LRs. In general, coalescing two live ranges either decreases the degrees of their neighbors or leaves them unchanged; it cannot increase their degrees.

The membership test should use the bit-matrix for efficiency.

To perform coalescing, the allocator walks the list of copies from *Build Graph* and inspects each operation, $LR_i \Rightarrow LR_j$. If $(LR_i, LR_j) \notin E$, then LR_i and LR_j do not interfere and the allocator combines them, eliminates the copy, and updates I to reflect the new, combined LR_{ij}. The allocator can conservatively update I by moving each edge from LR_i and LR_j to LR_{ij}, eliminating duplicates. This update is not precise, but it lets the allocator continue coalescing.

In practice, allocators coalesce every live range that they can, given the interferences in I. Then, they rewrite the code to reflect the revised LRs and eliminate the coalesced copies. Next, they rebuild I and try again to coalesce copies. This process typically halts after a couple of rounds of coalescing.

The example illustrates the imprecise nature of this conservative update to the graph. The update moves the edge (LR_b, LR_c) from LR_b to LR_{ab}, when, in fact, LR_{ab} and LR_c do not interfere. Rebuilding the graph from the transformed code yields the precise interference graph, without (LR_{ab}, LR_c). At that point, the allocator can coalesce LR_{ab} and LR_c.

If the allocator can coalesce LR_i with either LR_j or LR_k, choosing to form LR_{ij} may prevent a subsequent coalesce with LR_k, or vice versa. Thus, the order of coalescing matters. In principle, the compiler should coalesce the most frequently executed copies first. Thus, the allocator might coalesce copies in order by the estimated execution frequency of the block that contains the copy. To implement this, the allocator can consider the basic blocks in order from most deeply nested to least deeply nested.

In practice, the cost of building the interference graph for the first round of coalescing dominates the overall cost of the graph-coloring allocator. Subsequent passes through the build-coalesce loop process a smaller graph and, therefore, run more quickly. To reduce the cost of coalescing, the compiler can build a subset of the interference graph—one that only includes live ranges involved in a copy operation.

This strategy applies a lesson from semipruned SSA form: only include the names that matter.

13.4.4 Estimate Global Spill Costs

When a register allocator discovers that it cannot keep all of the live ranges in registers, it must select an LR to spill. Typically, the allocator uses some carefully designed metric to rank the choices and picks the LR that its metric suggests is the best spill candidate. The local allocator used the distance to the LR's next use, which works well in a single-block context. In the global allocator, the metric incorporates an estimate of the runtime costs that will be incurred by spilling and restoring a particular LR.

To compute the estimated spill costs for an LR, the allocator must examine each definition and use in the LR. At each definition, it estimates the cost of a spill after the definition and multiplies that number by the estimated execution frequency of the block that contains the definition. At each use, it estimates the cost of a restore before the use and multiplies that number by the estimated execution frequency of the block that contains the use. It sums together the estimated costs for each definition and use in the LR to produce a single number. This number becomes the spill cost for the LR.

Of course, the actual computation is more complex than the preceding explanation suggests. At a given definition or use of an LR, the value might be any of dirty, clean, or rematerializable (see Section 13.2.3). Individual definitions and uses within an LR can have different classifications, so the allocator must perform enough analysis to classify each reference in the LR. That classification determines the cost to spill or restore that reference.

The precise execution count of a block is difficult to determine. Fortunately, relative execution frequencies are sufficient to guide spill decisions; the allocator needs to know that one reference is likely to execute much more often than another. Thus, the allocator derives, for each block, a number that indicates its relative execution frequency. Those frequencies apply uniformly to each reference in the block.

The allocator could compute spill costs on demand—when it needs to make a spill decision. If it finds a *k*-coloring without any spills, an on-demand cost computation would reduce overall allocation time. If the allocator must spill frequently, a batch cost computation would, most likely, be faster than an on-demand computation.

Fig. 13.7 suggests that the allocator should perform the cost computation before it tries to color the graph. The allocator can defer the computation until the first time that it needs to spill. If the allocator does not need to spill, it avoids the overhead of computing spill costs; if it does spill, it computes spill costs for a smaller set of LRs.

Accounting for Execution Frequencies

Using the 10^d estimator can introduce a problem with integer overflow in the spill cost computation.

Many compiler writers have discovered this issue experimentally. Deeply nested loops may need floating-point spill costs.

To compute spill costs, the allocator needs an estimate of the execution frequency for each basic block. The compiler can derive these estimates from profile data or from heuristics. Many compilers assume that each loop executes 10 times, which creates a weight of 10^d for a block nested inside d loops. This assumption assigns a weight of 10 to a block inside one loop, 100 to a block inside two nested loops, and so on. An unpredictable `if–then–else` would decrease the estimated frequency by half. In practice, these estimates create a large enough bias to encourage spilling LRs in outer loops rather than those in inner loops.

Negative Spill Costs

A live range that contains a load, a store, and no other uses should receive a negative spill cost if the load and store refer to the same address. (Optimization can create such live ranges; for example, if the use were optimized away and the store resulted from a procedure call rather than the definition of a new value.) Sometimes, spilling a live range may eliminate copy operations with a higher cost than the spill operations; such a live range also has a negative cost. Any live range with a negative spill cost should be spilled, since doing so decreases demand for registers *and* removes operations from the code.

Infinite Spill Costs

```
      LRi ← ···
Mem[LRj] ← LRi
```

LR_i has spill cost ∞

Some live ranges are so short that spilling them does not help. Consider the short LR shown in the left margin. If the allocator tries to spill LR_i, it will insert a store after the definition and a load before the use, creating two new LRs. Neither of these new LRs uses fewer registers than the original LR, so the spill produces no benefit. The allocator should assign the original LR a spill cost of infinity, ensuring that the allocator does not try to spill it. In general, an LR should have infinite spill cost if no other LR ends between its definitions and its uses. This condition stipulates that availability of registers does not change between the definitions and uses.

```
initialize the stack and the spill list to empty
while ($N \neq \emptyset$) do
    if $\exists$ n $\in N$ with n° < $k$ then
        node ← n
    else node ← n picked by the spill metric
    remove node and its edges from $I$
    push node onto stack
```

(a) The Algorithm for *Simplify*

```
while (stack $\neq \emptyset$) do
    node ← pop(stack)
    insert node and its edges into $I$
    assign node a color not used
        by any of its neighbors
```

(b) The Algorithm for *Select*

■ **FIGURE 13.11** *Simplify* and *Select*: The Core Coloring Algorithm.

Infinite-cost live ranges present a code-shape challenge to the compiler. If the code contains more than $k-1$ nested infinite-cost LRs, and no LR ends in this region, then the infinite-cost LRs form an uncolorable clique in the interference graph. While such a situation is unusual, we have seen it arise in practice. The register allocator cannot fix this problem; the compiler writer must simply ensure that the allocator does not receive such code.

13.4.5 Color the Graph

The global allocator colors the graph in a two-step process. The first step, called *Simplify*, computes an order in which to attempt the coloring. The second step, called *Select*, considers each node, in order, and tries to assign it a color from its set of k colors.

To color the graph, the allocator relies on a simple observation:

> *If a node has fewer than k neighbors, then it must receive a color, independent of the colors assigned to its neighbors.*

Thus, any node n with degree less than k, denoted $n^\circ < k$, is trivial to color. The allocator first tries to color those nodes that are hard to color; it defers trivially colored nodes until after the difficult nodes have been colored.

Simplify

To compute an order for coloring, the allocator finds trivially colored nodes and removes them from the graph. It records the order of removal by pushing the nodes onto a stack as they are removed. The act of removing a node and its edges from the graph lowers the degree of all its neighbors. Fig. 13.11(a) shows the algorithm.

As nodes are removed from the graph, the allocator must preserve both the node and its edges for subsequent reinsertion in *Select*. The allocator can either build a structure to record them, or it can add a mark to each edge and each node indicating whether or not it is active.

Simplify uses two distinct mechanisms to select the node to remove next. If there exists a node n with $n^\circ < k$, the allocator chooses that node. Because these nodes are trivially colored, the order in which they are removed does not matter. If all remaining nodes are constrained, with degree $\geq k$, then the allocator picks a node to remove based on its *spill metric*. Any node n removed by this process has $n^\circ \geq k$; thus, it may not receive a color during the assignment phase. The loop halts when the graph is empty. At that point, the stack contains all the nodes in order of removal.

Spill metric
a heuristic used to select an LR to spill

Select

To color the graph, the allocator rebuilds the interference graph in the reverse of the removal order. It repeatedly pops a node n from the stack, inserts n and its edges back into I, and picks a color for n that is distinct from n's neighbors. Fig. 13.11(b) shows the algorithm.

In our experience, the order in which the allocator considers colors has little practical impact.

To select a color for node n, the allocator tallies the colors of n's neighbors in the current graph and assigns n an unused color. It can search the set of colors in a consistent order, or it can assign colors in a round-robin fashion. If no color remains for n, it is left uncolored.

When the stack is empty, I has been rebuilt. If every node has a color, the allocator rewrites the code, replacing LR names with PR names, and returns. If any nodes remain uncolored, the allocator spills the corresponding LRs. The allocator passes a list of the uncolored LRs to *Insert Spills*, which adds the spills and restores to the code. *Insert Spills* then restarts the allocator on the revised code. The process repeats until every node in I receives a color. Typically, the allocator finds a coloring and halts in a couple of trips around the large loop in Fig. 13.7.

Why Does This Work?

The global allocator inserts each node back into the graph from which it was removed. If the reduction algorithm removes the node for LR_n from I because $n^\circ < k$, then it reinserts LR_n into a graph in which $n^\circ < k$ and node n is trivially colored.

The only way that a node n can fail to receive a color is if n was removed from I using the spill metric. *Select* reinserts such a node into a graph in which $n^\circ \geq k$. Notice, however, that this condition is a statement about degree in the graph, rather than a statement about the availability of colors.

If node n's neighbors use all k colors, then the allocator finds no color for n. If, instead, they use fewer than k colors, then the allocator finds a color for n. In practice, a node n often has multiple neighbors that use the same color. Thus, *Select* often finds colors for some of these constrained nodes.

UPDATING THE INTERFERENCE GRAPH

Both coalescing and spilling change the set of nodes and edges in the interference graph. In each case, the graph must be updated before allocation can proceed.

The global coloring allocator uses a conservative update after each coalesce; that update also triggers another iteration around the *Build–Coalesce* loop in Fig. 13.7. It obtains precision in the graph by rebuilding it from scratch.

The allocator defers spill insertion until the end of the *Simplify–Select* process; it then inserts all of the spill code and triggers another iteration of the *Build–Coalesce–Spill Costs–Color* loop. Again, it obtains precision by rebuilding the graph.

If the allocator could update the graph precisely, it could eliminate both of the cycles shown in Fig. 13.7. Coalescing could complete in a single pass. It could insert spill code incrementally when it discovered an uncolored node; the updated graph would correctly reflect interferences for color selection.

Better incremental updates can reduce allocation time. A precise update would produce the same allocation as the original allocator, within variance caused by changes in the order of decisions. An imprecise but conservative update could produce faster allocations, albeit with some potential decrease in code quality from the imprecision in the graph. DasGupta

Simplify determines the order in which *Select* tries to color nodes. This order plays a critical role in determining which nodes receive colors. For a node n removed from the graph because $n° < k$, the order is unimportant with respect to the nodes that remain in the graph. The order may be important with respect to nodes already on the stack; after all, n may have been constrained until some of those earlier nodes were removed. For nodes removed from the graph using the spill metric, the order is crucial. The else clause in Fig. 13.11(a) executes only when every remaining node has degree $\geq k$. Thus, the nodes that remain in the graph at that point are in more heavily connected subgraphs of I.

The order of the constrained nodes is determined by the *spill metric*. The original coloring allocator picked a node that minimized the ratio of $cost \div degree$, where $cost$ is the estimated spill cost and $degree$ is the node's degree in the current graph. This metric balances between spill cost and the number of nodes whose degree will decrease.

The original global coloring allocator appeared in IBM's PL.8 compiler.

Other spill metrics have been tried. Several metrics are variations on $cost \div degree$, including $cost \div degree^2$, $cost \div area$ and $cost \div area^2$, where the *area* of an LR is defined as the sum of MAXLIVE taken over all the instructions that lie within the LR. These metrics try to balance the cost of spilling a specific LR against the extent to which that spill makes it easier to color the rest of the graph. Straight *cost* has been tried; it focuses on runtime speed. In code-space sensitive applications, a metric of *total spill operations* can drive down the code-space growth from spilling.

In practice, no single heuristic dominates the others. Since coloring is fast relative to building *I*, the allocator can color *I* with several different spill metrics and keep the best result.

13.4.6 **Insert Spill and Restore Code**

The spill code created by a global register allocator is no more complex than the spill code inserted in the local allocator. *Insert Spills* receives a list of LRs that did not receive a color. For each LR, it inserts the appropriate code after each definition and before each use.

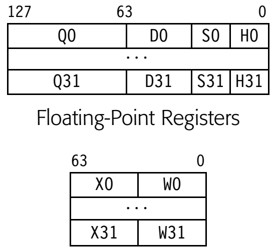
The same complexities that arose in the local allocator apply in the global case. The allocator should recognize the distinction between dirty values, clean values, and rematerializable values. In practice, it becomes more complex to recognize whether a value is dirty, clean, or rematerializable in the global scope.

The global allocator applies a *spill everywhere* discipline. An uncolored live range is spilled at every definition point and restored at every use point. In practice, a spilled LR often has definitions and uses that occur in regions where PRs are available. Several authors have looked at techniques to loosen the spill everywhere discipline so as to keep spilled LRs in PRs in regions of low register pressure (see the discussions in Section 13.5).

13.4.7 **Handling Overlapping Register Classes**

In practice, the register allocator must deal with the idiosyncratic properties of the target machine's register set and its calling convention. This reality constrains both allocation and assignment.

For example, on the ARM A-64, the four floating-point registers `Q1`, `D1`, `S1`, and `H1` all share space in a single PR, as shown in the margin. Thus, if the compiler allocates `D1` to hold LR$_i$, `Q1`, `S1`, and `H1` are unavailable while LR$_i$ is live. Similar restrictions arise with the overlapped general purpose registers, such as the pair `X3` and `W3`.



Floating-Point Registers

General Purpose Registers

Register Names on the ARM A-64

To understand how overlapping register classes affect the structure of a register allocator, consider how the local allocator might be modified to handle the ARM A-64 general purpose registers.

- The algorithm, as presented, uses one attribute to describe an LR, its virtual register number. With overlapping classes, such as Xi and Wi, each LR also needs an attribute to describe its class.
- The stack of free registers should use names drawn from one of the two sets of names, Xi or Wi. The state, allocated or free, of a coresident pair such as X0 and W0 is identical.
- The mappings *VRToPR*, *PRToVR*, and *PRNU* can also remain single-valued. If VR_i has an allocated register, *VRToPR* will map the VR's *num* field to a register number and its *class* field will indicate whether to use the X name or the W name.

Because the local algorithm has a simple way of modeling the status of the PRs, the extensions are straightforward.

To handle a more complex situation, such as the EAX register on the IA-32, the local allocator would need more extensive modifications. Use of EAX requires the entire register and precludes simultaneous use of AH or AL. Similarly, use of either AH or AL precludes simultaneous use of EAX. However, the allocator can use both AL and AH at the same time. Similar idiosyncratic rules apply to the other overlapping names, shown in the margin and in Fig. 13.2(b).

31	15	7 0
EAX	AH	AL
ECX	CH	CL
EDX	DH	DL
EBX	BH	BL
EBP	BP	
ESP	SP	
ESI	SI	
EDI	DI	

Intel IA-32 Register Names

The global graph-coloring allocators have more complex models for interference and register availability than the local allocator. To adapt them for fair use of overlapping register classes requires a more involved approach.

Describing Register Classes

Before we can describe a systematic way to handle allocation and assignment for multiple register classes, we need a systematic way to describe those classes. The compiler can represent the members of each class with a set. From Fig. 13.2(a), the ARM A-64 has six classes:

Q: {Q0, Q1, ..., Q31}	D: {D0, D1, ..., D31}	S: {S0, S1, ..., S31}
H: {H0, H1, ..., H31}	X: {X0, X1, ..., X31}	W: {W0, W1, ..., W31}

Thus, *class*(D3) = {D0, D1, ..., D31}, the set of 64-bit floating-point registers.

The simplest scheme to describe overlap between register classes is to introduce a function, *alias*(*r*). For a register name *r*, *alias*(*r*) maps *r* to the set of register names that occupy physical space that intersects with *r*'s space. In the ARM A-64, *alias*(W1) = {X1} and *alias*(S2) = {Q2, D2, H2}. Similarly, in IA-32, *alias*(AH) = {EAX}, *alias*(AL) = {EAX}, and *alias*(EAX) = {AL, AH}. Because AL and AH occupy disjoint space, they are not aliases of each other.

The compiler can compute the information that it will need for allocation and assignment from the *class* and *alias* relationships.

Coloring with Overlapping Classes

The presence of overlapping register classes complicates each of coloring, assignment, and coalescing.

Coloring

The graph-coloring allocator approximates colorability by comparing a node's degree against the number of available colors. If $n^\circ < k$, the number of available registers, then *Simplify* categorizes n as trivially colored, removes n from the graph, and pushes n onto the ordering stack. If the graph contains no trivially colored node, *Simplify* chooses the next node for removal using its spill metric.

The presence of multiple register classes means that k may vary across classes. For the node n that represents LR_{n}, $k = |\,class(\mathrm{LR}_{\mathrm{n}})\,|$.

The presence of overlapping register classes further complicates the approximation of colorability. If the LR's class has no aliases, then simple arithmetic applies; a single neighbor reduces the supply of possible registers by one. If the LR's class has aliases—that is, register classes overlap—then it may be the case that a single neighbor can reduce the supply of possible registers by more than one.

- In IA-32, EAX removes both AH and AL; it reduces the pool of 8-bit registers by two. The relationship is not symmetric; use of AL does not preclude use of AH. Unaligned floating-point register pairs create a more general version of this problem.
- By contrast, the ARM A-64 ensures that each neighbor counts as one register. For example, W2 occupies the low-order 32 bits of an X2 register; no name exists for X2's high-order bits. Floating-point registers have the same property; only one name at each precision is associated with a given 128-bit register (Qi).

To extend *Simplify* to work fairly and correctly with overlapping register classes, we must replace $n^\circ < k$ with an estimator that conservatively and correctly estimates colorability in the more complex case of overlapping register classes. Rather than tallying n's neighbors, we must count those neighbors with which n competes for registers.

Smith, Ramsey, and Holloway describe an efficient estimator that provides a fair and correct estimate of colorability. Their allocator precomputes supporting data from the *class* and *alias* relationships and then estimates n's colorability based on *n*'s class and the registers assigned to its relevant neighbors.

Assignment

Traditional discussions of graph-coloring allocators assume that the assignment of specific registers to specific live ranges does not have a significant impact on colorability. The literature ignores the difference between choosing colors "round-robin" and "first-in, first-out," except in unusual cases, such as biased coloring (see Section 13.5.1).

With overlapping register classes, some register choices can tie up more than one register. In IA-32, using EAX reduces the supply of eight bit registers by two, AH and AL, rather than one. Again, unaligned floating-point register pairs create a more general version of the problem. Just as one register assignment can conflict with multiple others, so too can one assignment alter the available incremental choices.

Consider looking for a single eight bit register on IA-32. If the available options were AL and CL, but AH was occupied and CH was not, then choosing AL might introduce fewer constraints. Because EAX already conflicts with AH, the choice of AL does not reduce the set of available E registers. By contrast, choosing CL would make ECX unavailable. Overlapping register classes complicate assignment enough to suggest that the allocator should choose registers with a more complex mechanism than the first-in, first-out stack from Section 13.3.

Coalescing

The compiler should only coalesce two LRs, LR_i and LR_j, if the resulting live range has a feasible register class. If both are general purpose registers, for example, then the combination works. If *class*(LR_i) contains only PR_2 and *class*(LR_j) contains only PR_4, then the allocator must recognize that the combined LR_{ij} would be overconstrained (see the further discussion in Section 13.5.1).

Coloring with Disjoint Classes

If the architecture contains sets of classes that are disjoint, the compiler can allocate them separately. For example, most processors provide separate resources for general purpose registers and floating-point registers. Thus, allocation of LR_i to a floating-point register has no direct impact on allocation in the general purpose register set. Because spills of floating-point values may create values that need general purpose registers, the floating-point allocation should precede the general purpose allocation.

If the allocator builds separate graphs for disjoint subclasses, it can reduce the number of nodes in the interference graph, which can yield significant compile-time savings, particularly during *Build Graph*.

Forcing Specific Register Placement

The allocator must handle operations that require placement of live ranges in specific PRs. These constraints may be either restrictions (LR_i must be in PR_j) or exclusions (LR_i cannot be in PR_k).

These constraints arise from several sources. The linkage convention dictates the placement of values that are passed in registers; these constraints include the ARP, some or all of the actual parameters, and the return value. Some hardware operations may require their operands in particular registers; for example, the 16-bit signed multiply on the IA-32 always reads an argument from AX and writes to both AX and DX.

On IA-32, AX describes a 16-bit register that overlaps AH and AL. Similarly, DX overlaps DH and DL.

The register class mechanism creates a simple way to handle such restrictions. The compiler writer creates a small register class for this purpose and attaches that class to the appropriate LRs. The coloring mechanism handles the rest of the details.

To handle exclusions, the compiler writer can build an exclusion set, again, a list of PRs, and attach it to specific LRs. The coloring mechanism can test prospective choices against the exclusion set. For example, between the code that saves the caller-saves registers and the code that restores them, the allocator should not use the caller-saves registers to hold anything other than a temporary value. A simple exclusion set will ensure this safe behavior.

SECTION REVIEW

Global register allocators operate over entire procedures. The presence of control flow makes global allocation more complex than local allocation. Most global allocators operate on the graph coloring paradigm. The allocator builds a graph that represents interferences between live ranges, and then it tries to find a coloring that fits into the available registers.

This section describes a global allocator that uses a precise interference graph and careful spill cost estimates. The precise interference graph enables a powerful copy-coalescing phase. The allocator spills with a simple greedy selection-heuristic and a spill-everywhere discipline. These choices lead to an efficient implementation.

REVIEW QUESTIONS

1. *Simplify* always removes trivially colored nodes ($n^{\circ} < k$) before it removes any constrained node ($n^{\circ} \geq k$). This suggests that it only spills a node that has at least k neighbors that are, themselves, constrained. Sketch a version of *Simplify* that uses this more precise criterion. How does its compile-time cost compare to the original algorithm? Do you expect it to produce different results?
2. The global allocator chooses a value to spill by finding the LR that minimizes some metric, such as *spill cost* ÷ *degree*. When the algorithm runs, it sometimes must choose several live ranges to spill before it makes any other live range unconstrained. Explain how this situation can happen. Can you envision a spill metric that avoids this problem?

13.5 ADVANCED TOPICS

Because the cost of a misstep during register allocation can be high, algorithms for register allocation have received a great deal of attention. Many variations on the global coloring allocator have been described in the literature and implemented in practice. Sections 13.5.1 and 13.5.2 describe other approaches to coalescing and spilling, respectively. Section 13.5.3 presents three different formulations of live ranges; each of these leads to a distinctly different allocator.

13.5.1 Variations on Coalescing

The coalescing algorithm presented earlier combines live ranges without regard to the colorability of the resulting live range. Several more conservative approaches have been proposed in the literature.

Conservative and Iterated Coalescing

Coalescing has both positive and negative effects. As mentioned earlier, coalescing LR_i and LR_j can reduce the degree of other LRs that interfere with both of them. However, $LR_{ij}^{\circ} \geq MAX(LR_i^{\circ}, LR_j^{\circ})$. If both LR_i and LR_j are trivially colored and $LR_{ij}^{\circ} \geq k$, then coalescing LR_i and LR_j increases the number of constrained LRs in the graph, which may or may not make the graph harder to color without spilling.

Conservative coalescing
The allocator only coalesces $LR_i \Rightarrow LR_j$ if the resulting LR does not make the graph harder to color.

Conservative coalescing attempts to limit the negative side effects of coalescing by only combining LR_i and LR_j if the result does not make the interference graph harder to color. Taken literally, this statement suggests the following condition:

> *Either* $LR^{\circ}_{ij} \leq MAX(LR^{\circ}_{i},LR^{\circ}_{j})$ *or* LR_{ij} has fewer than *k* neighbors with degree > *k*.

This condition is subtle. If one of LR_i or LR_j already has significant degree and coalescing LR_i and LR_j produces an LR with the same degree, then the result is no harder to color than the original graph. In fact, the coalesce would lower the degree of any LR that interfered with both LR_i and LR_j.

Comparisons against *k* must use the appropriate value for *class*(LR_i) and *class*(LR_j).

The second condition specifies that LR_{ij} should have the property that *Simplify* and *Select* will find a color for LR_{ij}. Say the allocator can coalesce LR_i and LR_j to create LR_{ij}. If LR_{ij} has degree greater than the two LRs that it replaces, but will still color, then the allocator can combine LR_i and LR_j. (The coalesce is still conservative.)

Conservative coalescing is attractive precisely because it cannot make the coloring problem worse. It does, however, prevent the compiler from coalescing some copies. Since degree is a loose upper bound on colorability, conservative coalescing may prevent some beneficial combinations and, thus, produce more spills than unconstrained coalescing.

Biased Coloring

Biased coloring
If LR_i and LR_j are connected by a copy, the allocator tries to assign them the same color.

Another way to coalesce copies without making the graph harder to color is to bias the choice of specific colors. *Biased coloring* defers coalescing into *Select*; it changes the color selection process. In picking a color for LR_i, it first tries colors that have been assigned to live ranges connected to LR_j by a copy operation. If it can assign LR_i a color already assigned to LR_j, then a copy from LR_i to LR_j, or from LR_j to LR_i, is redundant and the allocator can eliminate the copy operation.

To make this process efficient, the allocator can build, for each LR, a list of the other LRs to which it is connected by a copy. *Select* can then use these *partner lists* to quickly determine if some available color would allow the LR to combine with one of its partners. With a careful implementation, biased coloring adds little or no cost to the color selection process.

Iterated Coalescing

Iterated coalescing
The allocator repeats conservative coalescing before it decides to spill an LR.

In an allocator that uses conservative coalescing, some copies will remain uncoalesced because the resulting LR would have high degree. *Iterated coalescing* addresses this problem by attempting to coalesce, conservatively, before deciding to spill. *Simplify* removes nodes from the graph until no trivially colored node remains. At that point it repeats the coalescing phase. Copies that did not coalesce in the earlier graphs may coalesce in the reduced graph. If coalescing creates more trivially colored nodes, *Simplify* continues by removing those nodes. If not, it selects spill candidates from the graph until it creates one or more trivially colored nodes.

13.5.2 Variations on Spilling

The allocator described in Section 13.4 uses a "spill everywhere" discipline. In practice, an allocator can do a more precise job of spilling to relieve pressure in regions of high demand for registers. This observation has led to several interesting improvements on the spill-everywhere allocator.

Spilling Partial Live Ranges

The global allocator, as described, spills entire live ranges. This strategy can lead to overspilling if the demand for registers is low through most of the live range and high in a small region. More sophisticated spilling techniques find the regions where spilling a live range is productive—that is, the spill frees a register in a region where a register is truly needed. The global allocator can achieve similar results by spilling only in the region where interference occurs. One technique, called *interference-region spilling*, identifies a set of live ranges that interfere in the region of high demand and spills them only in that region. The allocator can estimate the costs of several spilling strategies for the interference region and compare those costs against the standard spill-everywhere approach. This kind of estimated-cost competition has been shown to improve overall allocation.

Clean Spilling

When the global allocator spills some LR_i, it inserts a spill after every definition and a restore before every use. If LR_i has multiple uses in a block where register pressure is low, a careful implementation can keep the value of LR_i in a register for its live subrange in that block. This improvement, sometimes called *clean spilling*, tries to ensure that a given LR is only restored once in a given block.

A variation on this idea would use a more general postpass over the allocated code to recognize regions where free registers are available and promote spilled values back into registers in those regions. This approach has been called *register scavenging*.

Rematerialization

Some values cost less to recompute than to spill. For example, small integer constants should be recreated with a load immediate rather than being retrieved from memory with a load. The allocator can recognize such values and rematerialize them rather than spill them.

Modifying a global graph-coloring allocator to perform rematerialization takes several small changes. The allocator must identify and tag SSA names that can be rematerialized. For example, any operation whose arguments are always available is a candidate. It can propagate these rematerialization tags over the code using a variant of the SSCP algorithm for constant-propagation described in Chapter 9. In forming live ranges, the allocator should only combine SSA names that have identical rematerialization tags.

The compiler writer must make the spill-cost estimation handle rematerialization tags correctly, so that these values have accurate spill-cost estimates. The spill-code insertion process must also examine the tags and generate the appropriate lightweight spills for rematerializable values. Finally, the allocator should use conservative coalescing to avoid prematurely combining live ranges with distinct rematerialization tags.

Live-Range Splitting

Spill code insertion changes both the code and the coloring problem. An uncolored LR is broken into a series of tiny LRs, one at each definition or use. The allocator can use a similar effect to improve allocation; it can deliberately split high-degree LRs in ways that either improve colorability or localize spilling.

Live-range splitting harnesses three distinct effects. If the split LRs have lower degrees than the original one, they may be easier to color—possibly even unconstrained. If some of the split LRs have high degree and, therefore, spill, then splitting may let the allocator avoid spilling other parts of the LR that have lower degree. Finally, splitting introduces spills at the points where the LR is broken. Careful selection of the split points can control the placement of some spill code—for example, encouraging spill code that lies outside of loops rather than inside of them.

Many approaches to splitting have been tried. One early coloring allocator broke uncolored LRs into block-sized LRs and then coalesced them back together when the coalesce did not make allocation harder, similar to conservative coalescing. Several approaches that use properties of the control-flow graph to choose split points have been tried. Results can be inconsistent; the underlying problems are still NP-complete.

Two particular techniques show promise. A method called *zero-cost splitting* capitalizes on nops in the instruction schedule to split LRs and improve both allocation and scheduling. A technique called *passive splitting* uses a directed interference graph to choose which LRs to split and where to split them; it decides between splitting and spilling based on the estimated costs of each alternative.

Implementing Splitting

The mechanics of introducing splits into a live range can be tricky. Briggs suggested a separate *split* operation that had the same behavior as a copy. His allocator used aggressive coalescing on copy operations. After the copies had been coalesced, it used conservative coalescing on the splits.

Promotion of Ambiguous Values

In code that makes heavy use of ambiguous values, whether derived from source-language pointers, array references, or object references whose class cannot be determined at compile time, the allocator's inability to keep such values in registers is a serious performance issue. To improve allocation of ambiguous values, several systems have included transformations that rewrite the code to keep unambiguous values in scalar local variables, even when their "natural" home is inside an array element or a pointer-based structure.

- Scalar replacement uses array-subscript analysis to identify reuse of array-element values and to introduce scalar temporary variables that hold reused values.
- Register promotion uses data-flow analysis of pointer values to find pointer-based values that can safely reside in a register throughout a loop nest. It rewrites the code to keep the value in a local scalar variable.

Both of these transformations encode the results of analysis into the shape of the code and make it obvious to the register allocator that these values can be kept in registers.

Promotion can increase the demand for registers. In fact, promoting too many values can produce spill code whose cost is greater than that of the memory operations that the transformation tries to avoid. Ideally, the promotion technique should use a measure of register pressure to help decide which values to promote. Unfortunately, good estimators for register pressure are hard to construct.

13.5.3 Other Forms of Live Ranges

The allocator in Section 13.4 operates over maximal-sized live ranges. Other allocators have used different notions of a live range, which changes both the allocator and the resulting allocation. These changes produce both beneficial and detrimental effects.

Shorter live ranges produce, in some cases, interference graphs that contain more trivially colored nodes. Consider a value that is live in one block with register pressure greater than k and in many blocks where demand is less than k. With maximal-sized LRs, the entire LR is nontrivial to color; with shorter LRs, some of these LRs may be trivially colored. This effect can lead to better register use in the areas of low pressure. On the downside, the shorter LRs still represent a single value. Thus, they must connect through copy operations or memory operations, which themselves have a cost.

Maximal-sized live ranges can produce general graphs. More precisely, for any graph, we can construct a procedure whose interference graph is isomorphic to that graph. Restricting the form of LRs can restrict the form of the interference graph. The following subsections describe three alternative formulations for live ranges; they each provide a high-level description of the allocators that result from these different formulations.

The Chapter Notes give references for the reader interested in a more detailed treatment of any of these allocators.

Each of these allocators represents a different point in the design space. Changing the definition of a live range affects both the precision of the interference graph and the cost of allocation. The tradeoffs are not straightforward, in large part because the underlying problems remain NP-complete and the allocators compute a quick approximation to the optimal solution.

Allocation Based on SSA Names

The interference graphs that result from maximal-sized live ranges in programs are general graphs. For general graphs, the problem of finding a k-coloring is NP-complete. There are, however, classes of graphs for which k-coloring can be done in polynomial time.

Chordal graph
a graph in which every cycle of more than three nodes has a *chord*—an edge that joins two nodes that are not adjacent in the cycle

In particular, the optimal coloring of a chordal graph can be found in $\mathbf{O}(|N| + |E|)$ time. The optimal coloring may use fewer colors, and thus fewer registers, than the greedy heuristic approach shown in Section 13.4.5. Of course, if the optimal coloring needs more than k colors, the allocator will still need to spill.

If the compiler treats every distinct SSA-name as a live range, then the resulting interference graph is a chordal graph. This observation sparked interest in global register allocation over the SSA-form of the code. An SSA-based allocator may find allocations that use fewer registers than the allocations found by the global coloring allocator.

If the graph needs more than k colors, the allocator still must spill one or more values. While SSA form does not lower the complexity of spill choice, it may offer some benefits. Global live ranges tend to have longer lifetimes than SSA names, which are broken by ϕ-functions at appropriate places in the code, such as loop headers and blocks that follow loops. These breaks give the allocator the chance to spill values over smaller regions than it may have with global live ranges.

Unfortunately, SSA-based allocation leaves the code in SSA form. The allocator, or a postpass, must translate out of SSA form, with all of the complications discussed in Section 9.3.5. That translation may increase demand for registers. An SSA-based allocator must be prepared to handle this situation.

If out-of-SSA translation needs to break a cycle of copies, it will require an extra register to do so.

Equally important, that translation inserts copy operations into the code; some of those copies may be extraneous. The allocator cannot coalesce away copies that implement the flow of values corresponding to a ϕ-function; to do so would destroy the chordal property of the graph. Thus, an SSA-based allocator would probably use a coalescing algorithm that does not use the interference graph. Several strong algorithms exist.

It is difficult to assess the relative merits of an SSA-based allocator and an allocator based on maximal-sized live ranges. The SSA-based allocator has the potential to obtain a better coloring than the traditional allocator, but it does so on a different graph. Both allocators must address the problems of spill choice and spill placement, which may contribute more to performance than the actual coloring. The two allocators use different techniques for copy coalescing. As with any register allocator, the actual implementation details will matter.

Allocation Based on Linear Intervals

The live ranges used in local allocation form an interval graph. We can compute the minimal coloring of an interval graph in linear time. A family of allocators called *linear scan* allocators capitalize on this observation; these allocators are efficient in terms of compile time.

Interval graph
a graph that depicts the intersections of intervals on a line

An interval interference graph has a node for each interval and an edge between two nodes if their intervals intersect.

Linear scan allocators ignore control flow and treat the entire procedure as a linear list of operations. The allocator represents the LR of a value v as an interval $[i, j]$ that contains all of the operations where v is live. That is, i is less than or equal to the ordinal number of the first operation where v is live and j is greater than or equal to the ordinal number of the last operation where v is live. As a result, the interference graph is an interval graph.

The interval $[i, j]$ may contain operations and blocks that would not be in the LR_V that the global allocator would construct. Thus, it can overestimate the precise live range.

To start, the allocator computes live information and builds a set of intervals to represent the values. It sorts the intervals into increasing order by the ordinal number of their first operations. At that point, it applies a version of the local allocation algorithm from Section 13.3. Values are allocated to free registers if possible; if no register is available, the allocator spills the LR whose interval has the highest ordinal number for its last operation.

The linear scan algorithm approximates the behavior of the local allocator. When the allocator needs to spill, it chooses the LR with the largest distance to the end of the interval (rather than distance to next use). It uses a spill-everywhere heuristic. These changes undoubtedly affect allocation; how they affect allocation is unclear.

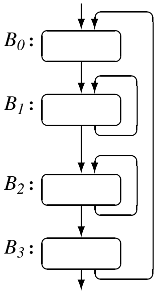
The linear scan allocator can coalesce a copy that is both the end of one LR and the start of another. This heuristic combines fewer LRs than the global coloring allocator might coalesce—an unavoidable side effect of using an implicit and approximate interference graph.

Live range splitting is a second attractive extension to linear scan. Breaking long LRs into shorter LRs can reduce MAXLIVE and allow the allocator to produce allocations with less spill code. To implement live range splitting, the compiler writer would need heuristics to select which LRs the allocator should split and where those splits should occur. Choosing the best set of splits is, undoubtedly, a hard problem.

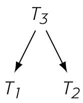
Linear scan allocators are an important tool for the compiler writer. Their efficiency makes them attractive for just-in-time compilers (see Chapter 14) and for small procedures where MAXLIVE $< k$. If they can allocate a procedure without spilling, then the allocation is, effectively, good enough.

Allocation Based on Hierarchical Coloring

The global allocator either assigns an LR to a register for its entire life, or it spills that LR at each of its definitions and uses. The hierarchical allocator takes ideas from live-range splitting and incorporates them into the way it treats live ranges. These modifications give the allocator a degree of control over the granularity and location of spilling.



CFG for a Loop Nest



Tile Tree for the CFG

In this scheme, the allocator imposes a hierarchical model on the nodes of the CFG. In the model, a *tile* represents one or more CFG nodes and the flow between them. Tiles are chosen to encapsulate loops. In the CFG shown in the margin, tile T_1 consists of $\{B_1\}$. Tile T_2 consists of $\{B_2\}$. Tiles nest; thus, tile T_3 contains $\{B_0, T_1, T_2, B_3\}$. The tile tree in the margin captures this relationship; T_1 and T_2 are siblings, as well as direct descendants of T_3.

The hierarchical allocator performs control-flow analysis to discover loops and group blocks into tiles. To provide a concrete representation for the nesting among the tiles, it builds a *tile tree* in which subtiles are children of the tile that contains them.

Next, the hierarchical allocator performs a bottom-up walk over the tile tree. At each tile, T_i, it builds an interference graph for the tile, performs coalescing, attempts to color the graph, and inserts spill code as needed. When it finishes with T_i, the allocator constructs a summary tile to represent T_i during the allocation of T_i's parent. The summary tile takes on the LIVEIN and LIVEOUT properties of the region that it represents, as well as the aggregate details of allocation in the region—the number of allocated registers and any PR preferences.

Once all the tiles have been individually colored, the allocator makes a top-down pass over the tile tree to perform assignment—that is, to map the allocation onto PRs. This pass follows the basic form of the global allocator, but it pays particular attention to values that are live across a tile boundary.

The bottom-up allocation pass discovers LRs one tile at a time. This process splits values that are live across tile boundaries; the allocator introduces copy operations for those splits. The split points isolate spill decisions inside a tile from register pressure outside a tile, which tends to drive spills to the boundaries of high-pressure tiles.

Cross-tile connections between live ranges become copy operations. The allocator uses a preferencing mechanism similar to biased coloring to remove these copies where practical (see Section 13.5.1). The same mechanism lets the allocator model requirements for a specific PR.

Of course, the allocator could run a postallocation coalescing pass over the allocated code.

Experiments suggest that the hierarchical allocator, with its shorter live ranges, produced slightly better allocations than a straightforward implementation of the global coloring allocator. Those same measurements showed that the allocator itself used more compile time than did the baseline global coloring allocator. The extra overhead of repeated allocation steps appears to overcome the asymptotic advantage of building smaller graphs.

13.6 SUMMARY AND PERSPECTIVE

Because register allocation is an important part of a modern compiler, it has received much attention in the literature. Strong techniques exist for both local and global allocation. Because many of the underlying problems are NP-hard, the solutions tend to be sensitive to small decisions, such as how ties between identically ranked choices are broken.

Progress in register allocation has come from the use of paradigms that provide intellectual leverage on the problem. Thus, graph-coloring allocators have been popular, not because register allocation is identical to graph coloring, but rather because coloring captures some of the critical aspects of the global allocation problem. In fact, many of the improvements to coloring allocators have come from attacking the points where the coloring paradigm does not accurately reflect the underlying problem, such as better cost models and improved methods for live-range splitting. In effect, these improvements have made the paradigm more closely fit the real problem.

CHAPTER NOTES

Register allocation dates to the earliest compilers. Backus reports that Best invented the algorithm from Section 13.3 in the mid-1950s for the original FORTRAN compiler [27,28]. Best's algorithm has been rediscovered and reused in many contexts [39,127,191,254]. It best-known incarnation is as Belady's offline page-replacement algorithm, MIN [39]. Horwitz [208] and Kennedy [225] both describe the complications created by clean and dirty values. Liberatore et al. suggest spilling clean values before dirty values as a compromise [254].

The connection between graph coloring and storage-allocation problems was suggested by Lavrov [250] in 1961; the Alpha project used coloring to pack data into memory [151,152]. Schwartz describes early algorithms by Ershov and by Cocke [320] that focus on using fewer colors and ignore spilling. The first complete graph-coloring allocator was built by Chaitin et al. for IBM's PL.8 compiler [80–82].

The global allocator in Section 13.4 follows Chaitin's plan with Briggs' modifications [57,58,62]. It uses Chaitin's definition of interference and the algorithms for building the interference graph, for coalescing, and for handling spills. Briggs added an SSA-based algorithm for live range construction, an improved coloring heuristic, and several schemes for live-range splitting [57].

The treatment of register classes derives from Smith, Ramsey, and Holloway [331]. Chaitin, Nickerson, and Briggs all discuss achieving some of the same goals by adding edges to the interference graph to model specific assignment constraints [60,82,284].

The notion of coloring disjoint subgraphs independently follows from Smith, Ramsey, and Holloway. Earlier, Gupta, Soffa, and Steele suggested partitioning the graph into independent graphs using clique separators [184] and Harvey proposed splitting it between general purpose and floating-point registers [111].

Many improvements to the basic Chaitin-Briggs scheme have appeared in the literature and in practice. These include stronger coalescing methods [168,289], better methods for spilling [40,41], register scavenging [193], rematerialization of simple values [61], and live-range splitting [107,116, 244]. Register promotion has been proposed as a preallocation transformation that rewrites the code to increase the set of values that can be kept in a register [73,77,258,261,315]. DasGupta proposed a precise incremental update for coalescing and spilling, as well as a faster but somewhat lossy update [124]. Harvey looked at coloring spill locations to reduce spill memory requirements [193].

The SSA-based allocators developed from the independent work of several authors [64,186,292]. Both Hack and Bouchez built on the original observation with in-depth treatments [53,185]. Linear scan allocation was proposed by Poletto and Sarkar [296]. The hierarchical coloring scheme is due to Koblenz and Callahan [75,106].

EXERCISES

Section 13.3

1. Apply the local allocation algorithm to the following ILOC basic block. Assume that r_{arp} and r_i are live on entry to the block and that the target machine has four physical registers.

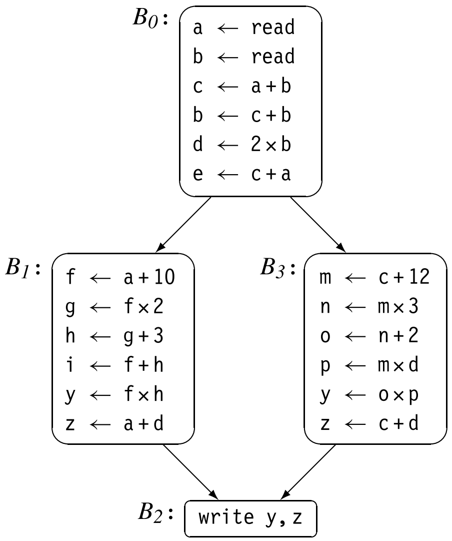
```
loadAI  r_arp, 12  ⇒ r_a
loadAI  r_arp, 16  ⇒ r_b
add     r_i, r_a   ⇒ r_c
sub     r_b, r_i   ⇒ r_d
mult    r_c, r_d   ⇒ r_e
multI   r_b, 2     ⇒ r_f
add     r_e, r_f   ⇒ r_g
storeAI r_g        ⇒ r_arp, 8
```

2. Consider the control-flow graph shown in Fig. 13.12. Assume that `read` returns a value from external media and that `write` transmits a value to external media.

   a. Compute the LiveIn and LiveOut sets for each block.

   b. Assuming three physical registers, apply the local allocation algorithm to blocks B_0, B_1 and B_3. If block b defines a name n and $n \in$ LiveOut(b), the allocator must store n back to memory so that its value is available in subsequent blocks. Similarly, if block b uses a name n before any local definition of n, it must load n's value from memory. Show the resulting code, including all loads and stores.

   c. Suggest a scheme that would allow some of the values in LiveOut(B_0) to remain in registers, avoiding their initial loads in the successor blocks.



```
B0:
a ← read
b ← read
c ← a + b
b ← c + b
d ← 2 × b
e ← c + a
```

```
B1:
f ← a + 10
g ← f × 2
h ← g + 3
i ← f + h
y ← f × h
z ← a + d
```

```
B3:
m ← c + 12
n ← m × 3
o ← n + 2
p ← m × d
y ← o × p
z ← c + d
```

```
B2:
write y, z
```

■ **FIGURE 13.12** Control-Flow Graph for Exercise 2.

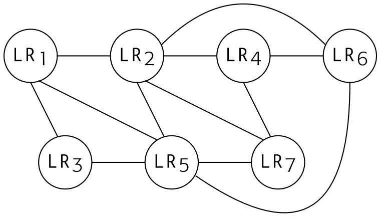
3. The *Build–Coalesce* loop tends to dominate the running time of the global allocator. One way to reduce this cost is to build an interference graph for *Coalesce* that only includes LRs that are involved in one or more copy operations.

   a. Sketch a method to build this reduced graph.

   b. Discuss the costs and benefits, assuming that graph construction takes $O(n^2)$ time where n is the number of nodes in the graph. Assume that coalescing reduces the number of live ranges.

Section 13.4

4. Consider the following interference graph:



Assume that the target machine has three physical registers and that each live range has an estimated spill cost of 10.

   a. Apply the allocation algorithm from Section 13.4 to the graph. Which live ranges are spilled? Which are colored?

b. Does the choice of spill node make a difference?

c. Chaitin's allocator used a different approach to *Simplify* and *Select* than does the allocator from Section 13.4. In his allocator, *Simplify* removed nodes from the graph in the same way and in the same order as in the global allocator. *Simplify* build a list of these node that were removed with the spill heurisitic. At the end of *Simplify*, if that "*spill list*" was nonempty, the allocator immediately called *Insert Spills* to spill all the nodes on the list. It then restarted the allocation process on this modified code.

What happens when you apply Chaitin's algorithm to the example interference graph? Does this algorithmic change produce the same results or different results?

5. After register allocation, examination of the code may discover some stretches of the code where some registers are free. In the global coloring allocator, this occurs because of detailed shortcomings in the way that live ranges are spilled.

   a. Explain how this situation can arise.

   b. How might the compiler discover if this situation occurs and where it occurs?

   c. What might be done to use these unused registers, both within the global framework and outside of it?

6. When the global allocator discovers that no color is available for a live range, LR_i, it spills or splits that live range. As an alternative, it might attempt to recolor one or more of LR_i's neighbors. Consider the case where $(LR_i,LR_j) \in I$ and $(LR_i,LR_k) \in I$, but $(LR_j,LR_k) \notin I$. If LR_j and LR_k have already been colored, and they have different colors, the allocator might be able to recolor one of them to the other's color, freeing a color for LR_i.

   a. Sketch an algorithm that discovers if a legal and productive recoloring exists for LR_i.

   b. What is the impact of your technique on the asymptotic complexity of the register allocator?

   c. If the allocator cannot recolor LR_k to the same color as LR_j because one of LR_k's neighbors has the same color as LR_j, should the allocator consider recursively recoloring LR_k's neighbors? Explain your rationale.

7. The description of the global allocator suggests inserting spill code for *every* definition and use in the spilled live range.

   If a given block has one or more free registers, spilling a live range multiple times in that block is wasteful. Suggest an improvement to the spill mechanism in the global allocator that avoids this problem.

8. Consider a procedure that consists of a single basic block with 10,000 operations and maximum register pressure of $k + 15$.

   a. How does the spilling behavior of the local allocator compare with that of the global allocator on such a block?

   b. Which allocator do you expect to produce less spill code?

Chapter 14

Runtime Optimization

ABSTRACT

Runtime optimization has become an important technique for the implementation of many programming languages. The move from ahead-of-time compilation to runtime optimization lets the language runtime and its compilers capitalize on facts that are not known until runtime. If these facts enable specialization of the code, such as folding an invariant value, avoiding type conversion code, or replacing a virtual function call with a direct call, then the profit from use of runtime information can be substantial.

This chapter explores the major issues that arise in the design and implementation of a runtime optimizer. It describes the workings of a hot-trace optimizer and a method-level optimizer; both are inspired by successful real systems. The chapter lays out some of the tradeoffs that arise in the design and implementation of these systems.

KEYWORDS

Runtime Compilation, Just-in-Time Compilation, Dynamic Optimization

14.1 INTRODUCTION

Many programming languages include features that make it difficult to produce high-quality code at compile time. These features include late binding, dynamic loading of both declarations and code (classes in JAVA), and various kinds of polymorphism. A classic compiler, sometimes called an *ahead-of-time* compiler (AOT), can generate code for these features. In many cases, however, it does not have sufficient knowledge to optimize the code well. Thus, the AOT compiler must emit the generic code that will work in any situation, rather than the tailored code that it might generate with more precise information.

Runtime optimization
code optimization applied at runtime

For some problems, the necessary information might be available at link time, or at class-load time in JAVA. For others, the information may not be known until runtime. In a language where such late-breaking information can have a significant performance impact, the system can defer optimization or translation until it has enough knowledge to produce efficient code.

Engineering a Compiler. https://doi.org/10.1016/B978-0-12-815412-0.00020-6

Compiler writers have applied this strategy, *runtime optimization* or *just-in-time compilation* (JIT), in a variety of contexts, ranging from early LISP systems through modern scripting languages. It has been used to build regular-expression search facilities and fast instruction-set emulators. This chapter describes the technical challenges that arise in runtime optimization and runtime translation, and shows how successful systems have addressed some of those problems.

Just-in-time compilers are, undoubtedly, the most heavily used compilers that the computer science community has built. Most web browsers include JITs for the scripting languages used in web sites. Runtime systems for languages such as JAVA routinely include a JIT that compiles the heavily used code. Because these systems compile the code every time it runs, they perform many more compilations than a traditional AOT compiler.

Conceptual Roadmap

Classic AOT compilers make all of their decisions based on the facts that they can derive from the source text of the program. Such compilers can generate highly efficient code for imperative languages with declarations. However, some languages include features that make it impossible for the compiler to know important facts until runtime. Such features include dynamic typing, some kinds of polymorphism, and an open class structure.

Runtime optimization involves a fundamental tradeoff between time spent compiling and code quality. The runtime optimizer examines the program's state to derive more precise information; it then uses that knowledge to specialize the executable code. Thus, to be effective, the runtime optimizer must derive useful information. It must improve runtime performance enough to compensate for the added costs of optimization and code generation. The compiler writer, therefore, has a strong incentive to use methods that are efficient, effective, and broadly applicable.

A Few Words About Time

Runtime optimization adds a new layer of complexity to our reasoning about time. These techniques intermix compile time with runtime and incur compile-time costs every time a program executes.

JIT time
We refer to the time when the runtime optimizer or the just-in-time compiler is, itself, executing as JIT time.

At a conceptual level, the distinction between compile time and runtime remains. The runtime optimizer plans runtime behavior and runtime data structures, just as an AOT compiler would. It emits code to create and maintain the runtime environment. Thus, JIT-time activities are distinct from runtime activities. All of the reasoning about time from earlier chapters is relevant, even if the time frame when the activities occur has shifted.

To further complicate matters, some systems that use runtime optimization rely on an interpreter for their default execution mode. These systems interpret code until they discover a segment of code that should be compiled. At that point they compile and optimize the code; they then arrange for subsequent executions to use the compiled code for the segment. Such systems intermix interpretation, JIT compilation, and execution of compiled code.

A "segment" might be a block, a trace, a method, or multiple procedures.

Overview

To implement efficiently features such as late binding of names to types or classes, dynamic loading and linking of code, and polymorphism, compiler writers have turned to runtime optimization. A runtime optimizer can inspect the running program's state to discover information that was obscured or unavailable before runtime.

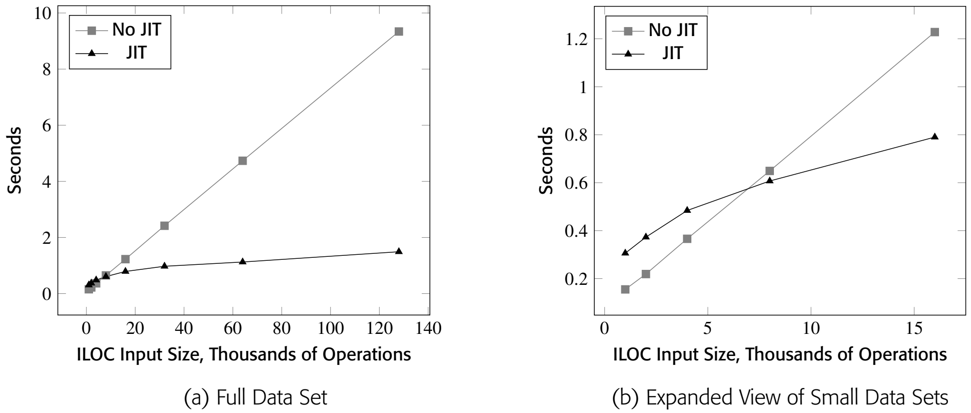
By runtime, the system mostly knows what code is included in the executable. Late bound names have been resolved. Data structures have been allocated, so their sizes are known. Objects have been instantiated, with full class information. Using facts from the program's runtime state, a compiler can specialize the code in ways that are not available to an AOT compiler.

Runtime compilation also provides natural mechanisms to deal with runtime changes in the program's source text (see Section 14.5.4).

Runtime compilation has a long history. McCarthy's early LISP systems compiled native code for new functions at runtime. Thompson's construction, which builds an NFA from a regular expression, was invented to compile an RE into native code inside the search command for the QED editor—one of the first well-known examples of a compiler that executed at runtime. Subsequent systems used these techniques for purposes that ranged from the implementation of dynamic object-oriented languages such as SMALLTALK-80 through code emulation for portability. The rise of the World Wide Web was, in part, predicated on widespread use of JAVA and JAVASCRIPT, both of which rely on runtime compilation for efficiency.

Runtime optimization presents the compiler writer with a novel set of challenges and opportunities. Time spent in the compiler increases the overall running time, so the JIT writer must balance JIT costs against expected improvement. Techniques that shift compilation away from infrequently executed, or cold, code and toward frequently executed, or hot, code can magnify any gain from optimization.

We use the term JIT to cover all runtime optimizers, whether their input is source code, as in McCarthy's early LISP systems; some high-level notation, as in Thompson's RE-to-native-code compiler; some intermediate form as in JAVA systems; or even native code, as in Dynamo. The digressions throughout this chapter will introduce some of these systems, to familiarize the reader with the long history and varied applications of these ideas.



■ **FIGURE 14.1** Scalability of a JAVA Application.

Impact of JIT Compilation

The data was gathered on OpenJDK version 1.8.0_292 running on an Intel ES2640 at 2.4GHz.

The input codes had uniform register pressure of 20 values. The allocator was allotted 15 registers.

JIT compilation can make a significant difference in the execution speed of an application. As a concrete example, Fig. 14.1 shows the running times of a JAVA implementation of the local register allocation algorithm from Section 13.3. Panel (a) shows the running time of the allocator on a series of blocks with 1,000 lines, 2,000 lines, 4,000 lines, and so on up to 128,000 lines of ILOC code. The gray line with square data points shows the running time with the JIT disabled; the black line with triangular data points shows the running time with the JIT enabled. Panel (b) zooms in on the startup behavior—that is, the smaller data sets.

These numbers are specific to this single application. Your experience will vary.

The JIT makes a factor of six difference on the largest data set; it more than compensates for the time spent in the JIT. Panel (a) shows the JIT's contribution to the code's performance. Panel (b) shows that VM-code emulation is actually faster on small data sets. Time spent in the JIT slows execution in the early stages of runtime; after roughly one-half second, the speed advantage of the compiled code outweighs the costs incurred by the JIT.

Roadmap

JIT design involves fundamental tradeoffs between the amount of work performed ahead of time, the amount of work performed in the JIT, and the improvement that JIT compilation achieves. As languages, architectures, and runtime techniques have changed, these tradeoffs have shifted. These tradeoffs will continue to shift and evolve as the community's experience with building and deploying JITs grows. Our techniques and our understanding will almost certainly improve, but the fundamental tradeoff of efficiency against effectiveness will remain.

This chapter provides a snapshot of the state of the field at the time of publication. Section 14.2 describes four major issues that play important roles in shaping the structure of a JIT-enabled system. The next two sections present high-level sketches for two JITs that sit at different points in the design space. Section 14.3 describes a hot-trace optimizer while Section 14.4 describes a hot-method optimizer; both designs are modeled after successful systems. The Advanced Topics section explores several other issues that arise in the design and construction of practical JIT-based systems.

14.2 BACKGROUND

Runtime optimization has emerged as a technology that lets the runtime system adapt the executable code more closely to the context in which it executes. In particular, by deferring optimization until the compiler has more complete knowledge about types, constant values, and runtime behavior (e.g., profile information), a JIT compiler can eliminate some of the overhead introduced by language features such as object-orientation, dynamic typing, and late binding.

Success in runtime optimization, however, requires attention to both the efficiency and the effectiveness of the JIT. The fundamental principle of runtime optimization is

> *A runtime compiler must save more cycles than it uses.*

If the runtime compiler fails to meet this constraint, then it actually slows down the application's execution.

This critical constraint shapes both the JIT and the runtime system with which it interacts. It places a premium on efficiency in the compiler itself. Because compile time now adds to running time, the JIT implementation's efficiency directly affects the application's running time. Both the scope and ambition of the JIT matter; both asymptotic complexity and actual runtime overhead matter. Equally important, the scheme that chooses which code segments to optimize has a direct impact on the total cost of running an application.

This constraint also places a premium on the effectiveness of each algorithm that the JIT employs. The compiler writer must focus on techniques that are both widely applicable and routinely profitable. The JIT should apply those techniques to regions where opportunities are likely and where those improvements pay off well. A successful JIT improves the code's running time often enough that the end users view time spent in the JIT as worthwhile.

REGULAR EXPRESSION SEARCH IN THE QED EDITOR

Ken Thompson built a regular-expression (RE) search facility into the QED editor in the late 1960s. This search command was an early JIT compiler, invoked under the user's direction. When the user entered an RE, the editor invoked the JIT to create native code for the IBM 7094. The editor then invoked the native code to perform the search. After the search, it discarded the code.

The JIT was a compiler. It first parsed the RE to check its syntax. Next, it converted the RE to a postfix notation. Finally, it generated native code to perform the search. The JIT's code generator used the method now known as Thompson's construction to build, implicitly, an NFA (see Section 2.4.2). The generated code simulated that NFA. It used search to avoid introducing duplicate terms that would cause exponential growth in the runtime state.

The QED search command added a powerful capability to a text editor that ran on a 0.35 MIP processor with 32 KB of RAM. This early use of JIT technology created a responsive tool that ran in this extremely constrained environment.

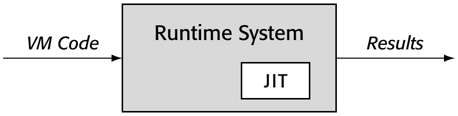
This situation differs from that which occurs in an AOT compiler. Compiler writers assume that the code produced by an AOT compiler executes, on average, many times per compilation. Thus, the cost of optimization is a small concern. AOT compilers apply a variety of transformations that range from broadly applicable methods such as value numbering to highly specific ones such as strength reduction. They employ techniques that produce many small improvements and others that produce a few large improvements. An AOT compiler wins by accumulating the improvements from a suite of optimizations, used at every applicable point in the code. The end user is largely insulated from the cost of compilation and optimization.

To recap, the constraints in a JIT mean that the JIT writer must choose transformations well, implement them carefully, and apply them to regions that execute frequently. Fig. 14.1 demonstrates the improvement from JIT compilation with the HotSpot Server Compiler. In that test, HotSpot produced significant improvements for codes that ran for more than one-half of a second. Careful attention to both costs and benefits allows this JIT to play a critical role in JAVA's runtime performance.

14.2.1 **Execution Model**

The choice of an *execution model* has a large impact on the shape of a runtime optimization system. It affects the speed of baseline execution. It affects the amount of compilation that the system must perform and, therefore, the cumulative overhead of optimization. It also affects the complexity of the implementation.

A runtime optimization system can be complex. It takes, as input, code for some virtual machine (VM) code. The VM code might be code for an abstract machine such as the JAVA VM (JVM) or the Smalltalk-80 VM. In other systems, the VM code is native machine code. As output, the runtime optimizer produces the results of program execution.



The runtime system can produce results by executing native code, by interpreting VM code, or by JIT compiling VM code to native code and running it. The relationship between the JIT, the code, and the rest of the runtime system determines the mode of execution. Does the system execute, by default, native code or VM code? Either option has strengths and weaknesses.

The difference between these modes is largely transparent to the user.

- native-code execution usually implies JIT compilation before execution, unless the VM code *is* native code.
- VM-code execution usually implies interpretation at some level. The code is compact; it can be more abstract than native code.

Native-code execution is, almost always, faster than VM-code execution. Native code relies on hardware to implement the fetch-decode-execute cycle, while VM emulation implements those functions in software. Lower cost per operation turns into a significant performance advantage for any nontrivial execution.

On the other hand, VM-code systems may have lower startup costs, since the system does not need to compile anything before it starts to execute the application. This leads to faster execution for short-running programs, as shown in Fig. 14.1(b). For procedures that are short or rarely executed, VM-code emulation may cost less than JIT compilation plus native-code execution.

A VM-code system can defer scanning and parsing a procedure until it is called. The savings in startup time can be substantial.

The introduction of a JIT to a VM-code system typically creates a mixed-mode platform that executes both VM code and native code. A mixed-mode system may need to represent critical data structures, such as activation records, in both the format specified for the VM and the format supported by the native ISA. The dual representations may introduce translation between VM-code structures and native-code structures; those translations, in turn, will incur runtime costs.

The Deutsch-Schiffman SMALLTALK-80 system used three formats for an AR. It translated between formats based on whether or not the code accessed the AR as data.

ADAPTIVE FORTRAN

Adaptive Fortran was a runtime optimizer for FORTRAN IV built by Hansen as part of the work for his 1974 dissertation. He used it to explore both the practicality and the profitability of runtime optimization. Adaptive Fortran introduced many ideas that are found in modern systems.

The system used a fast ahead-of-time compiler to produce an IR version of the program; the IR was grouped into basic blocks. At runtime, the IR was interpreted until block execution counts indicated that the block could benefit from optimization. (The AOT compiler produced block-specific, optimization-specific thresholds based on block length, nesting depth, and the cost of the JIT optimization.)

Guided by the execution counts and thresholds, a supervisor invoked a JIT to optimize blocks and their surrounding context. The use of multiple block-specific thresholds led to an effect of progressive optimization—more optimizations were applied to blocks that accounted for a larger share of the running time.

One key optimization, which Hansen called *fusion*, aggregated together multiple blocks to group loops and loop nests into *segments*. This strategy allowed Adaptive Fortran to apply loop-based optimizations such as code motion.

The alternative, native-code execution, distributes the costs in a different way. Such a system must compile all VM code to native code, either in an AOT compilation or at runtime. The AOT solution leads to fully general code and, thus, a higher price for nonoptimized native execution. The JIT solution leads to a system that performs more runtime compilation and incurs those costs on each execution.

There is no single best solution to these design questions. Instead, the compiler writer must weigh carefully a variety of tradeoffs and must implement the system with an eye toward both efficiency and effectiveness. Successful systems have been built at several points in this design space.

14.2.2 Compilation Triggers

The runtime system must decide when and where to invoke the JIT. This decision has a strong effect on overall performance because it governs how often the JIT runs and where the JIT focuses its efforts.

Runtime optimizers use JIT compilation in different ways. If the system JIT compiles *all* code, as happens in some native-code systems, then the trigger may be as simple as “compile each procedure before its first call.” If, instead, the system only compiles hot code, the trigger may require procedure-level or block-level profile data. Native-code environments and mixed-mode environments may employ different mechanisms to gather that profile data.

In a native-code environment, the compiler writer must choose between (1) a system that works from VM code and compiles that VM code to native code before it executes, or (2) a system that works from AOT-compiled native code and only invokes the JIT on frequently executed, or hot, code. The two approaches lead to distinctly different challenges.

VM-Code Execution

In a mixed-mode environment, the system can begin execution immediately and gather profile data to determine when to JIT compile code for native execution. These systems tend to trigger compilation based on profile data exceeding a preset threshold value above which the code is considered hot. This approach helps the system avoid spending JIT cycles on code that has little or no impact on performance.

Threshold values play a key role in determining overall runtime. Larger threshold values decrease the number of JIT compilations. At the same time, they increase the fraction of runtime spent in the VM-code emulator, which is typically slower than native-code execution. Varying the threshold values changes system behavior.

To obtain accurate profile data, a VM-code environment can instrument the application's branches and jumps. To limit the overhead of profile collection, these systems often limit the set of points where they collect data. For example, blocks that are the target of a backward branch are good candidates to profile because they are likely to be loop headers. Similarly, the block that starts a procedure's prolog code is an obvious candidate to profile. The system can obtain call-site specific data by instrumenting precall sequences. All of these metrics, and others, have been used in practical and effective systems.

Backward branch
In this context, a *backward* branch or jump targets an address smaller than the program counter.

Loop-closing branches are usually backward branches.

Native-Code Execution

If the system executes native code, it must compile each procedure before that code can run. The system can trigger compilation at load time, either in batch for the entire executable (Speed Doubler) or as modules are loaded (early versions of the V8 system for JAVASCRIPT). Alternatively, the system can trigger the compiler to translate each procedure the first time it runs. To achieve that effect, the system can link a stub in place of any yet-to-be-compiled procedure; the stub locates the VM code for the callee, JIT compiles and links it, and reexecutes the call.

SPEED DOUBLER

Speed Doubler was a commercial product from Connectix in the 1990s. It used load-time compilation to retarget legacy applications to a new ISA. Apple had recently migrated its Macintosh line of computers from the Motorola MC 68000 to the IBM POWER PC. Support for legacy applications was provided by an emulator built into the MacOS.

Speed Doubler was a load-time JIT that translated MC 68000 applications into native POWER PC code. By eliminating the emulation overhead, it provided a substantial speedup. When installed, it was inserted between the OS loader and the start of application execution. It did a quick translation, then branched to the application's startup code.

The initial version of Speed Doubler appeared to perform an instruction-by-instruction translation, which provided enough improvement to justify the product's name. Subsequent versions provided better runtime performance; we presume it was the result of better optimization and code generation.

Speed Doubler used native-code execution with a compile-on-load discipline to provide a simple and transparent mechanism to improve running times. Users perceived that JIT compilation cost significantly less than the speedups that it achieved, so the product was successful.

Load-time strategies must JIT-compile every procedure, whether or not it ever executes. Any delay from that initial compilation occurs as part of the application's startup. Compile-on-call shifts the cost of initial compilation later in execution. It avoids compiling code that never runs, but it does compile any code that runs, whether it is hot or cold.

Decreasing time in the JIT directly reduces elapsed execution time.

If the system starts from code compiled by an AOT compiler, it can avoid these startup compilations. The AOT compiler can insert the necessary code to gather profile data. It might also annotate the code with information that may help subsequent JIT compilations. A system that uses precompiled native code only needs to trigger the optimizer when it discovers that some code fragment is hot—that is, the code consumes enough runtime to justify the cost of JIT compiling it.

14.2.3 Granularity of Optimization

Runtime optimizers operate at different granularities. This decision, made at design time, has a strong impact on overall effectiveness because it determines the kinds of optimizations that the JIT can apply and the code size of the fragments that the JIT optimizes. Two particular optimization scopes have been used widely.

Hot Traces A trace optimizer watches runtime branches and jumps to discover hot traces. Once a trace's execution count exceeds the preset hot threshold, the system invokes the JIT to construct an optimized native-code implementation of the trace.

Trace optimizers perform local or regional optimization on the hot trace, followed by native-code generation including allocation and scheduling. Because a runtime trace may include calls and returns, this "regional" optimization can make improvements that would be considered interprocedural in an AOT compiler.

Trace (revisited)
A trace is an acyclic sequence of blocks. As used in runtime optimization, a trace can cross procedure-call boundaries.

Hot Methods A method optimizer finds procedures that account for a significant fraction of overall running time by monitoring various counters. These counters include call counts embedded in the prolog code, loop iteration counts collected before backward branches, and call-site specific data gathered in precall sequences. Once a method becomes hot, the system uses a JIT to compile optimized native code for the method.

Because it works on the entire method, the optimizer can perform nonlocal optimizations, such as code motion, regional instruction scheduling, dead-code elimination, global redundancy elimination, or strength reduction. Some method optimizers also perform inline substitution. They might pull inline the code for a frequently executed call in the hot method. If most calls to a hot method come from one call site, the optimizer might inline the callee into that caller.

The choice of granularity has a profound impact on both the cost of optimizations and the opportunities that the optimizer discovers.

- A trace optimizer might apply LVN to the entire trace to find redundancy, fold constants, and simplify identities. Most method optimizers use a global redundancy algorithm, which is more expensive but should find more opportunities for improvement.

Assume a trace that has one entry but might have premature exits.

- A trace optimizer might use a fast local register allocator like the algorithm from Section 13.3. By contrast, a method optimizer must deal with control flow, so it needs a global register allocator such as the coloring allocator or the linear scan allocator (see Sections 13.4 and 13.5.3).

Linear scan achieves some of the benefits of global allocation with lower cost than graph coloring.

Again, the tradeoff comes down to the cost of optimization against the total runtime improvement.

14.2.4 Sources of Improvement

A JIT can discover facts that are not known before runtime and use those facts to justify or inform optimization. These facts can include profile information, object types, data structure sizes, loop bounds, constant values or

THE DEUTSCH-SCHIFFMAN SMALLTALK-80 SYSTEM

The Deutsch-Schiffman implementation of Smalltalk-80 used JIT compilation to create a native-code environment on a system with a Motorola MC 68000-series processor. Smalltalk-80 was distributed as an image for the Smalltalk-80 virtual machine.

This system only executed native-code. The method lookup and dispatch mechanism invoked the JIT for any method that did not have a native-code body—a compile-on-call discipline.

The system gained most of its speed improvement from replacing VM emulation with native-code execution. It used a global method cache and was the first system to use inline method caches. The authors were careful about translating between VM structures and native-code structures, particularly activation records. The result was a system that was astonishing in its speed when compared to other contemporary Smalltalk-80 implementations on off-the-shelf hardware.

The system ran in a small-memory environment. (16MB of RAM was considered large at the time.) Because native code was larger than VM code by a factor of two to five, the system managed code space carefully. When the system needed to reclaim code space, it discarded native code rather than paging it to disk. This strategy, sometimes called *throw-away code generation*, was profitable because of the large performance differences between VM emulation and native-code execution, and between JIT compilation and paging to a remote disk (over 10 MBPS Ethernet).

types, and other system-specific facts. To the extent that these facts enable optimization that cannot be done in an AOT compiler, they help to justify runtime compilation.

In practice, runtime optimizers find improvement in a number of different ways. Among those ways are:

In the Deutsch-Schiffman system, native code was fast enough to compensate for the JIT costs.

- *Eliminate VM Overhead* If the JIT operates in a mixed-mode environment, the act of translation to native code decreases the emulation overhead. The native code replaces software emulation with hardware execution, which is almost always faster.

  Some early JITs, such as Thompson's JIT for regular expressions in the QED editor, performed minimal optimization. Their benefits came, almost entirely, from elimination of VM overhead.
- *Improve Code Layout* A trace optimizer naturally achieves improvements from code layout. As it creates a copy of the hot trace, the JIT places the blocks in sequential execution order, with some of the benefits ascribed to global code placement (see Section 8.6.2).

In the compiled copy of the hot trace, the JIT can make the on-trace path use the fall-through path at each conditional branch. At the same time, any end-of-block jumps in the trace become jumps to the next operation, so the JIT can simply remove them.

Dynamo, in particular, benefited from linearization of the traces.

- *Eliminate Redundancy* Most JITs perform redundancy elimination. A trace optimizer can apply the LVN or SVN algorithms, which also perform constant propagation and algebraic simplification. Both algorithms have $O(1)$ cost per operation.

  A method optimizer can apply DVNT or a data-flow technique such as lazy code motion or a global value-numbering algorithm to achieve similar benefits. The costs of these algorithms vary, as do the specific opportunities that they catch (see Sections 10.6 and 10.3.1).
- *Reduce Call Overhead* Inline substitution eliminates call overhead. A runtime optimizer can use profile data to identify call sites that it should inline. A trace optimizer can subsume a call or a return into a trace. A method optimizer can inline call sites into a hot method. It can also use profile data to decide whether or not to inline the hot method into one or more of its callers.
- *Tailor Code to the System* Because the results of JIT compilation are ephemeral—they are discarded at the end of the execution—the JIT can optimize the code for the specific processor model on which it will run.

  The JIT might tailor a compute-bound loop to the available SIMD hardware or the GPU. Its scheduler might benefit from model-specific facts such as the number of functional units and their operation latencies.

We do not know of a JIT that performs model-specific optimization.

For machine-dependent problems such as instruction scheduling, the benefits might be significant.

- *Capitalize on Runtime Information* Programs often contain facts that cannot be known until runtime. Of particular interest are constant, or unchanging, values. For example, loop bounds might be tied to the size of a data structure read from external media—read once and never changed during execution. The JIT can determine those values and use them to improve the code. For example, it might move range-checks out of a loop (see Section 7.3.3).

  In languages with late binding, type and class information may be difficult or impossible to discern in an AOT compiler. The JIT can use runtime knowledge about types and classes to tailor the compiled code to the runtime reality. In particular, it might convert a generic method dispatch to a class-specific call.

An AOT compiler might identify values that can impact JIT optimization and include methods that query those values.

JIT compilation can impose subtle constraints on optimization. For example, traditional AOT optimization often focuses on loops. Thus, techniques such as unrolling, loop-invariant code motion, and strength reduction have all proven important in the AOT model. Hot-trace optimizers that exclude cyclic paths cannot easily duplicate those effects.

THE DYNAMO HOT-TRACE OPTIMIZER

The Dynamo system was a native-code, hot-trace optimizer for Hewlett-Packard's PA-8000 systems. The system's fundamental premise was that it could efficiently identify and improve frequently executed traces while executing infrequently executed code in emulation.

To find hot traces, Dynamo counted the executions of blocks that were likely start-of-trace candidates. When a block's count crossed a preset threshold (50), the JIT would build a trace and optimize it. Subsequent executions of the trace ran the compiled code. The system maintained its own software-managed cache of compiled traces.

Dynamo achieved improvements from local and superlocal optimization, from improved code locality, from branch straightening, and from linking traces into larger fragments. Its traces could cross procedure-call boundaries, which allowed Dynamo to optimize interprocedural traces.

Dynamo showed that JIT compilation could be profitable, even in competition with code optimized by an AOT compiler. Subsequent work by others created a Dynamo-like system for the IA-32 ISA, called DynamoRIO.

Control-flow optimizations, such as unrolling or cloning, typically require a control-flow graph. It can be difficult to reconstruct a CFG from assembly code. If the code uses a *jump-to-register* operation (jump in ILOC), it may be difficult or impossible to know the actual target. In an IR version of the code, such branch targets can be recorded and analyzed. Even with *jump-to-label* operations (jumpI in ILOC), optimization may obfuscate the control-flow to the point where it is difficult or impossible to reconstruct. For example, Fig. 12.17 on page 656 shows a single-cycle, software-pipelined loop that begins with five *jump-to-label* operations; reconstructing the original loop from the CFG in Fig. 12.17(b) is a difficult problem.

ILOC includes the tbl pseudooperation to record and preserve this kind of knowledge.

14.2.5 Building a Runtime Optimizer

JIT construction is an exercise in engineering. It does not require new theories or algorithms. Rather, it requires careful design that focuses on efficiency and effectiveness, and implementation that focuses on minimizing actual costs. The success of a JIT-based system will depend on the cumulative impact of individual design decisions.

The rest of this chapter illustrates the kinds of tradeoffs that occur in a runtime optimizer. It examines two specific use cases: a hot-trace optimizer, in Section 14.3, and a hot-method optimizer, in Section 14.4. The hypothetical hot-trace optimizer draws heavily from the design of the Dynamo system.

The hot-method optimizer takes its inspiration from the original HotSpot Server Compiler and from the Deutsch-Schiffman SMALLTALK-80 system. Finally, Section 14.5 builds on these discussions to examine some of the more nuanced decisions that a JIT designer must make.

SECTION REVIEW

In JIT design, compiler writers must answer several critical questions. They must choose an execution model; will the system run unoptimized code in an emulator or as native code? They must choose a granularity for compilation; typical choices are traces and whole procedures (or methods). They must choose the compilation triggers that determine when the system will optimize (and reoptimize) code. Finally, compiler writers must understand what sources of improvement the JIT will target, and they must choose optimizations that help with those particular issues.

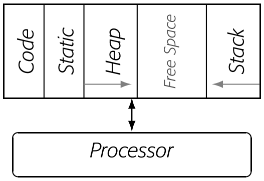
Throughout the design and implementation process, the compiler writer must weigh the tradeoffs between spending more time on JIT compilation and the resulting reduction of time spent executing the code. Each of these decisions can have a profound impact on the effectiveness of the overall system and the running time of an application program.

REVIEW QUESTIONS

1. In a system that executes native code by default, how might the system create the profile data that it needs? How might the system provide that data to the JIT?
2. Eliminating the overhead of VM execution is, potentially, a major source of improvement. In what situations might emulation be more efficient than JIT compilation to native code?

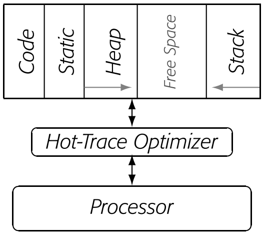
14.3 HOT-TRACE OPTIMIZATION

In the classic execution model for compiled code, the processor reads operations and data directly from the address space of the running process. The drawing labeled “Normal Execution” in the margin depicts this situation. (It is a simplified version of Fig. 5.15.) The fetch-decode-execute cycle uses the processor’s hardware.



Normal Execution

Conceptually, a native-code hot-trace optimizer sits between the executing process’ address space and the processor. It “monitors” execution until it has “enough” context to determine that some portion of the code is hot and should be optimized. At that point, it optimizes the code and ensures that future executions of the optimized sequence run the optimized copy rather than the original code. The margin drawing depicts that situation.



Execution Mediated by a Runtime Optimizer

The hot-trace optimizer has a difficult task. It must find hot traces. It must improve those traces enough to overcome the costs of finding and compiling them. For each cycle spent in the JIT, it must recover one or more cycles through optimization. In addition, if the process slows execution of the cold code, the optimized compiled code must also make up that deficit.

This section presents the design of a hot-trace optimizer. The design follows that of the Dynamo system built at Hewlett-Packard Research around the year 2000. It serves as both an introduction to the issues that arise and a concrete example to explore design tradeoffs.

Dynamo executed native code by emulation until it identified a hot trace. The emulator ran all of the cold code, gathered profile information, and identified the hot traces. Thus, emulated execution of the native code was slower than simply running that code on the hardware. The premise behind Dynamo was that improvement from optimizing hot traces could make up for both the emulation overhead and the JIT compilation costs.

These design concepts raise several critical issues. How should the system define a trace? How can it find the traces? How does it decide a trace is hot? Where do optimized traces live? How does emulated cold code link to the hot code and vice versa?

Trace-Entry Blocks

In Dynamo, trace profiling, trace identification, and linking hot and cold code all depend on the notion of a *trace-entry block*. Each trace starts with an entry block. A trace-entry block meets one of two simple criteria. Either it is the target of a backward branch or jump, or it is the target of an exit from an existing compiled trace.

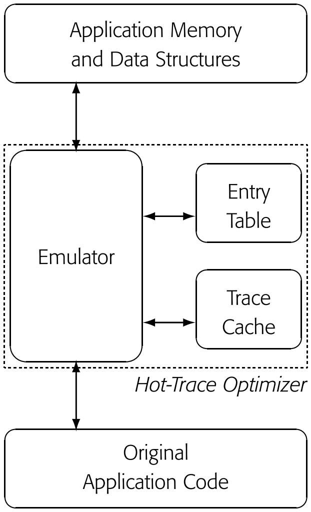
The first criterion selects blocks that are likely to be loop-header blocks. These blocks can be identified with an address comparison; if the target address is numerically smaller than the current program counter (PC), the target address designates a trace-entry block.

The second criterion selects blocks that may represent alternate paths through a loop. Any side exit from a trace becomes a trace-entry block. The JIT identifies these blocks as it compiles a hot trace.

To identify and profile traces, the trace optimizer finds trace-entry blocks and counts the number of times that they execute. Limiting the number of profiled blocks helps keep overhead low. As the optimizer discovers entry

```
L1: emulate code until a taken branch or jump
    lookup target address in the entry table
    if address is not in the table then
        if address < emulated PC then
            create table entry for address
            set target address' counter to 1
    else  // address is in the entry table
        if fragment has been compiled then
            jump to the compiled fragment
        bump the target address' counter
        if counter > hot threshold then
            build, compile, & execute the trace
    // if here, block runs in emulation mode
    set emulator's PC to target address
    jump back to L1
```

(a) High-level Algorithm



(b) Components of the Trace Optimizer

■ **FIGURE 14.2** Conceptual Structure of a Hot-Trace Optimizer.

blocks, it enters them into a table—the entry table. The table contains an execution count and a code pointer for each block. It is a critical data structure in the hot-trace optimizer.

14.3.1 **Flow of Execution**

Fig. 14.2(a) presents a high-level algorithm for the trace optimizer. The combination of the trace-entry table and the trace cache encapsulates the system's current state. The algorithm determines how to execute a block based on the presence or absence of that block in the entry table and the values of its execution counter and its code pointer.

Blocks run in emulation until they become part of a compiled hot trace. At that point, further executions of the compiled trace run the block as part of the optimized code. If control enters the block from another off-trace path, the block executes in emulation using the original code.

The critical set of decisions occurs when the emulator encounters a taken branch or a jump. (A jump is always taken.) At that point, the emulator looks for the target address in the trace entry table.

- If the target address is not in the table and that address is numerically smaller than the current PC, the system classifies the target address as a trace entry block. It creates an entry in the table and initializes the

The smaller target address means that this branch or jump is a backward branch.

entry's execution counter to one. It then sets the emulator's PC to the target address and continues emulation with the target block.

- If, instead, the target address already has a table entry and that entry has a valid code pointer, the emulator transfers control to the compiled code fragment. Depending on the emulator's implementation, discussed below, this transfer may require some brief setup code, similar to the precall sequence in a classic procedure linkage.

  The compiled trace is stored in the trace cache.

  Each exit path from the compiled trace either links to another compiled trace or ends with a short stub that sets the emulator's PC to the address of the next block and jumps directly back to the emulator—to label *L1* in Fig. 14.2(a).

  Section 14.3.2 discusses how the compiler can link compiled traces together.

- If the target address is in the table but has not yet been compiled, the system increments the target address' execution counter and tests it against the hot threshold. If the counter is less than or equal to the threshold, the system executes the target block by emulation. When the target address' execution counter crosses the threshold, the system builds an IR image of the hot trace, executes and compiles that trace, stores the code into the trace cache, and stores its code pointer into the appropriate slot in the trace entry table.

  On exit from the compiled trace, execution continues with the next block. Either the code links directly to another compiled trace or it uses a path-specific exit stub to start emulation with the next block.

The algorithm in Fig. 14.2(a) shows the block-by-block emulation, interspersed with execution of optimized traces. The emulator jumps into optimized code; optimized traces exit with code that sets the emulator's PC and jumps back to the emulator. The rest of this section explores the details in more depth.

Emulation

Following Dynamo, the trace optimizer executes cold code by emulation. The JIT-writer could implement the emulator as a full-fledged interpreter with a software fetch-decode-execute loop. That approach would require a simulated set of registers and code to transfer register values between simulated registers and physical registers on the transitions between emulated and compiled code. This transitional code might resemble parts of a standard linkage sequence.

Values that live in memory can use the same locations in both execution modes.

As an alternative, the system could "emulate" execution by running the original compiled code for the block and trapping execution on a taken branch or jump. If hardware support for that trap is not available, the system can store the original operation and replace it with an illegal instruction—a trick used in debuggers since the 1960s.

When the PC reaches the end of the block, the illegal instruction traps. The trap handler then follows the algorithm from Fig. 14.2(a), using the stored operation to determine the target address. In this approach, individual blocks execute from native code, which may be faster than a software fetch-decode-execute loop.

Building the Trace

When a trace-entry block counter exceeds the hot threshold, the system invokes the optimizer with the address of the entry block. The optimizer must then build a copy of the trace, optimize that copy, and enter the optimized fragment into the trace cache.

The optimizer leaves the original code intact and in-place so that other paths, such as side entries into the trace, can still use those blocks in emulation.

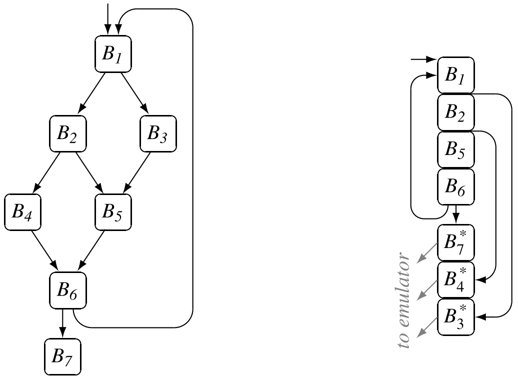
While the system knows that the entry block has run more than *threshold* times, it does not actually know which path or paths those executions took. Dynamo assumes that the current execution will follow the hot path. Thus, the optimizer starts with the entry block and executes the code in emulation until it reaches the end of the trace—a taken backward branch or a transfer to the entry of a compiled trace. Again, comparisons of runtime addresses identify these conditions.

As the optimizer executes the code, it copies each block into a buffer. At each taken branch or jump, it checks for the trace-ending conditions. An untaken branch denotes a side exit from the trace, so the optimizer records the target address so that it can link the side exit to the appropriate trace or exit stub. When it reaches the end of the trace, the optimizer has both executed the trace and built a linearized version of the code for the JIT to optimize.

Consider the example shown in Fig. 14.3(a). When the emulator sees B_1's counter cross the hot-threshold, it invokes the optimizer. The optimizer executes B_1 and copies each of its operations into the buffer. The next branch takes control to B_2; the emulator executes B_2 and adds it to the buffer. Next, control goes to B_5 followed by B_6. The branch at the end of B_6 goes back to B_1, terminating the trace. At this point, the buffer contains B_1, B_2, B_5, and B_6, as shown in panel (b).

The drawing assumes that each of the side exits leads to cold code. Thus, the JIT builds a stub to handle each side exit and the end-of-trace exit. The stub labeled B_{i}^* sets the emulator's PC to the start of block B_i and jumps to the emulator. The stub also provides a location where the optimizer can insert any code needed to interface the compiled code with the emulated code. Panel (b) shows stubs for B_3, B_4, and B_7.

The optimizer builds the trace based on the dynamic behavior of the executing code, which can produce complex effects. For example, a trace can



(a) Example CFG

(b) Trace Buffer for $\langle B_1, B_2, B_5, B_6 \rangle$

■ **FIGURE 14.3** Building a Trace.

extend through a procedure call and, with a simple callee, through a return. Because call and return are implemented with jumps rather than branches, they will not trigger the criteria for an exit block.

Optimizing the Trace

Once the optimizer has constructed a copy of the trace, it makes one or more passes over the trace to analyze and optimize the code. If the initial pass is a backward pass, the optimizer can collect LIVE information and other useful facts. From an optimization perspective, the trace resembles a single path through an extended basic block (see Section 8.5). In the example trace, an operation in B_6 can rely on facts derived from any of its predecessors, as they all must execute before control can reach this copy of B_6.

The mere act of trace construction should lead to some improvements in the code. The compiler can eliminate any on-trace jumps. For each early exit, the optimizer should make the on-trace path be the fall-through path. This linearization of the code should provide a minor performance improvement by eliminating some branch and jump latencies and by improving instruction-cache locality.

The compiler writer must choose the optimizations that the JIT will apply to the trace. Value numbering is an obvious choice; it eliminates redundancies, folds constants, and simplifies algebraic identities.

If the trace ends with a branch to its entry block, the optimizer can unroll this path through the loop. In a loop with control flow, the result may be a loop that is unrolled along some paths and not along others—a situation that does not arise in a traditional AOT optimizer.

Early exits from the trace introduce complications. The same compensation-code issues that arise in regional scheduling apply to code motion across early exits (e.g., at the ends of B_1 and B_2). If optimization moves an operation across an exit, it may need to insert code into the stub for that exit.

The optimizer can detect some instances of dead or partially dead code. Consider an operation that defines r_i. If r_i is redefined before its next on-trace use, then the original definition can be moved into the stubs for any early exits between the original definition and the redefinition. If it is not redefined but not used in the trace, it can be moved into the stubs for the early exits and into the final block of the trace.

Partially dead
An operation is *partially dead* at point p in the code if it is live on some paths that start at p and dead on others.

After optimization, the compiler should schedule operations and perform register allocation. Again, the local versions of these transformations can be applied, with compensation code at early exits.

Trace-Cache Size

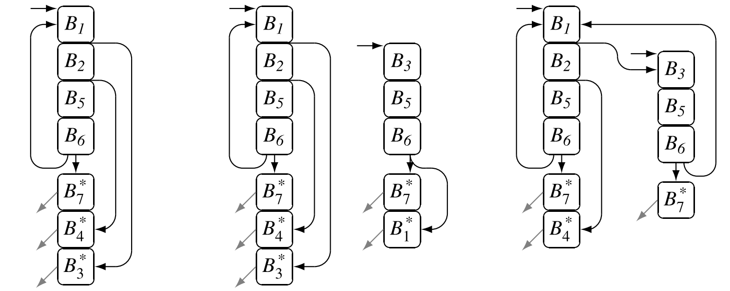
The size of the trace cache can affect performance. Size affects multiple aspects of trace-cache behavior, from memory locality to the costs of lookups and managing replacements. If the cache is too small, the JIT may discard fragments that are still hot, leading to lost performance and subsequent recompilations. If the cache is too large, it may retain code that has gone cold, hurting locality and raising lookup costs. Undoubtedly, compiler writers need to tune the trace-cache size to the specific system characteristics.

14.3.2 Linking Traces

One key to efficient execution is to recognize when other paths through the CFG become hot and to optimize them in a way that works well with the fragments already in the cache.

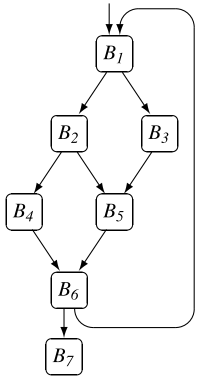
In the ongoing example, block B_1 became hot and the optimizer built a fragment for $\langle B_1, B_2, B_5, B_6 \rangle$, as shown in Fig. 14.4(a). The early exits to B_3 and B_4 then make those blocks into trace-entry blocks. If B_3 becomes hot, the optimizer will build a trace for it. The only trace it can build is $\langle B_3, B_5, B_6 \rangle$, as shown in panel (b).

If the optimizer maintains a small amount of information about trace entries and exits, it can link the two traces in panel (b) to create the code shown in panel (c). It can rewrite the branch to B_1^* as a direct jump to B_1. Similarly, it can rewrite the branch to B_3^* as a direct jump to B_3. The interlinked traces then create fast execution paths for both $\langle B_1, B_2, B_5, B_6 \rangle$ and $\langle B_1, B_3, B_5, B_6 \rangle$, as shown in panel (c). The exits to B_4 and B_7 still run though their respective stubs to the interpreter.



(a) Original Trace (b) Trace for $\langle B_3, B_5, B_6\rangle$ (c) After Trace Linking

■ **FIGURE 14.4** Adding a Second Trace.



Example CFG

If during optimization of $\langle B_1, B_2, B_5, B_6\rangle$, the JIT moved operations into B_3^*, then the process of linking would need to either (1) preserve B_3^* on the path from B_1 to B_3 or (2) prove that the operations in B_3^* are dead. With a small amount of context, such as the set of registers defined before use in the fragment, it could recognize dead compensation code in the stub.

Cross-linking in this way also addresses a weakness in the trace-construction heuristic. The trace builder assumed that the $(k+1)$st execution of B_1 took the hot path. Because the system only instruments trace header blocks, B_1's execution count could have accrued from multiple paths between B_1 and B_6. What happens if the $(k+1)$st execution takes the least hot of those paths?

With trace linking, the $(k+1)$st execution will build an optimized fragment. If that execution does not follow the hot path, then one or more of the early exits in the fragment will become hot; the optimizer will compile them and link them into the trace, capturing the hot path or paths. The optimizer will recover gracefully as it builds a linked set of traces.

Intermediate Entries to a Trace

In the example, when B_1 became hot, the system built an optimized trace for $\langle B_1, B_2, B_5, B_6\rangle$. When B_3 became hot, it optimized $\langle B_3, B_5, B_6\rangle$.

The algorithm, as explained, builds a single trace for $\langle B_1, B_2, B_5, B_6\rangle$ and ignores the intermediate entry to the trace from the edge (B_3, B_5). The system then executes the path $\langle B_3, B_5, B_6\rangle$ by emulation until B_3's counter triggers compilation of that path. This sequence of actions produces two copies of the code for B_5 and B_6, along with the extra JIT-time to optimize them.

Another way to handle the edge (B_3, B_5) would be to construct an intermediate entry into the trace $\langle B_1, B_2, B_5, B_6\rangle$. The trace-building algorithm, as explained, ignores these intermediate entry points, which simplifies record-keeping. If the emulator knew that B_5 was an intermediate entry point, it could split the trace on entry to B_5. It would build an optimized trace for $\langle B_1, B_2\rangle$ and another for $\langle B_5, B_6\rangle$. It would link the expected-case exit from $\langle B_1, B_2\rangle$ to the head of $\langle B_5, B_6\rangle$.

To implement trace splitting, the optimizer needs an efficient and effective mechanism to recognize an intermediate trace entry point—to distinguish, in the example, between B_4 and B_5. The hot-trace optimizer, as described, does not build an explicit representation of the CFG. One option might be for the AOT compiler to annotate the VM code with this information.

B_4 has one predecessor while B_5 has two.

Splitting $\langle B_1, B_2, B_5, B_6\rangle$ after B_2 may produce less efficient code than compiling the unsplit trace. Splitting the trace avoids compiling $\langle B_5, B_6\rangle$ a second time and storing the extra code in the code cache. It requires an extra slot in the entry block table. This tradeoff appears to be unavoidable. The best answer might well depend on the length of the common suffix of the two paths, which may be difficult to discern when compiling the first trace.

SECTION REVIEW

A hot-trace optimizer identifies frequently executed traces in the running code, optimizes them, and redirects future execution to the newly optimized code. It assumes that frequent execution in the past predicts frequent execution in the future and focuses the JIT's effort on such "hot" code. The acyclic nature of the traces leads to the use of local and superlocal optimizations. Those methods are fast and can capture many of the available opportunities.

The use of linked traces and interprocedural traces lets a hot-trace optimizer achieve a kind of partial optimization that an ahead-of-time compiler would not. The intent is to focus the JIT's effort where it should have maximum effect, and to limit its effort in regions where the expected impact is small.

REVIEW QUESTIONS

1. Once a trace entry block becomes hot, the optimizer chooses the rest of the trace based on the entry-block's next execution. Contrast this strategy with the trace-discovery algorithm used in trace-scheduling. How might the results of these two approaches differ?
2. Suppose the trace optimizer fills its trace cache and must evict some trace. What steps would be needed to revert a specific trace so that it executes by VM-code emulation?

14.4 HOT-METHOD OPTIMIZATION

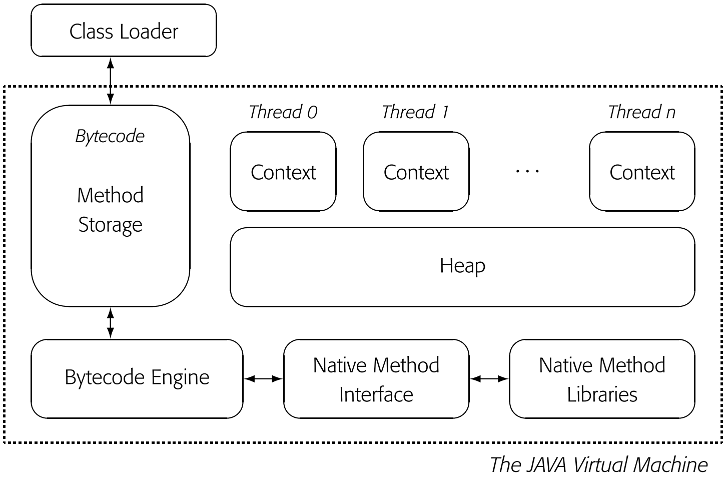
Method-level optimization presents a different set of challenges and tradeoffs than does trace-level optimization. To explore these issues, we will first consider a hot-method optimizer embedded in a JAVA environment. Our design is inspired by the original HotSpot Server Compiler (hereafter, HotSpot). The design is a mixed-mode environment that runs cold methods as JAVA bytecode and hot methods as native code. We finish this section with a discussion of the differences in a native-code hot-method optimizer.

14.4.1 Hot-Methods in a Mixed-Mode Environment

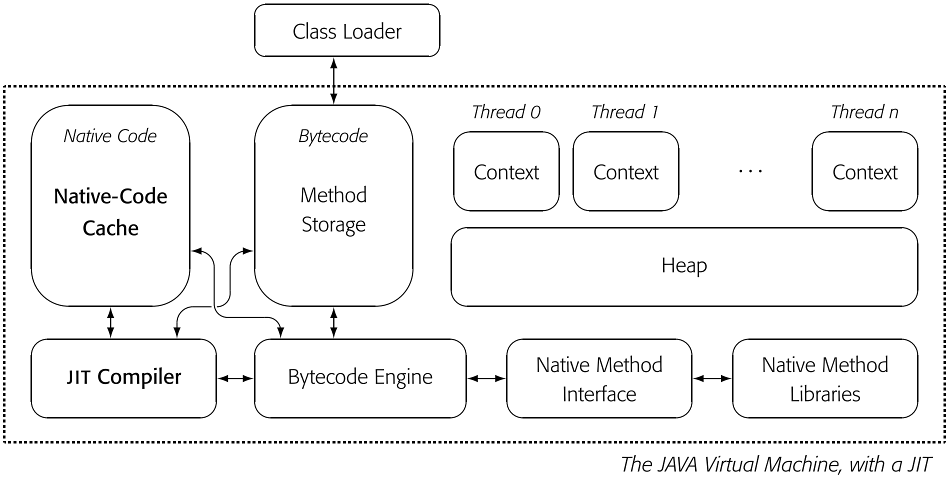
Fig. 14.5 shows an abstract view of the JAVA virtual machine or JVM. Classes and their associated methods are loaded into the environment by the Class Loader. Once stored in the VM, methods execute on an emulator—the figure's "Bytecode Engine." The JVM operates in a mixed-mode environment, with native-code implementations for many of the standard methods in system libraries.

To add a method-level JIT, the compiler writer must add several features to the JVM: the JIT itself, a software-managed cache for native-code method bodies, and appropriate interfaces. Fig. 14.6 shows these modifications.

From an execution standpoint, the presence of a JIT brings several changes. Cold code still executes via VM-code emulation; methods from native libraries still execute from native code. When the system decides that a method is hot, it JIT-compiles the VM code into native code and stores the



■ **FIGURE 14.5** The JAVA Runtime Environment.



■ **FIGURE 14.6** Adding a JIT to the JAVA Runtime Environment.

new code in its native-code cache. Subsequent calls to that method run from the native code, unless the system decides to revert the method to VM-code emulation (see the discussion of deoptimization on page 742).

The compiler writer must make several key decisions. The system needs a mechanism to decide which methods it will compile. The system needs a strategy to gather profile information efficiently. The compiler writer must decide whether the native code operates on the VM-code or the native-code versions of the various runtime structures, such as activation records. The compiler writer must design and implement the JIT, which is just an efficient and constrained compiler. Finally, the compiler writer must design and implement a mechanism to revert a method to VM-code emulation when the compiled method proves unsuitable. We will explore these issues in order.

Using native-code ARs may necessitate translation between native-code and VM-code formats.

The JAVA community often refers to ARs as "stack frames."

Trigger for Compilation

Conceptually, the hot-method optimizer should compile a method when that method consumes a significant fraction of the execution time. Finding a hot method that meets this criterion is harder than finding a hot trace, because the notion of a "significant fraction" of execution time is both imprecise and unknowable until the program terminates.

Thus, hot-method optimizers fall back on counters and thresholds to estimate a method's activity. This approach relies on the implicit assumption that a method that has consumed significant runtime will continue to consume significant runtime in the future. Our design, following HotSpot, will measure: (1) the number of times the method is called and (2) the number

Iteration can occur with either loops or recursion. The mechanism should catch either case.

THE HOTSPOT SERVER COMPILER

Around 2000, Sun Microsystems delivered a pair of JITs for its JAVA environment: one intended for client-side execution and the other for server-side execution. The original HotSpot Server Compiler employed more expensive and extensive techniques than did the client-side JIT. The HotSpot Server compiler was notable in that it used strong global optimization techniques and fit them into the time and space constraints of a JIT. The authors used an IR that explicitly represented both control flow and data flow [92]. The IR, in turn, facilitated redundancy elimination, constant propagation, and code motion. Sparsity in the IR helped make these optimizations fast.

The JIT employed a novel global scheduling algorithm and a full coloring allocator (see Section 13.4). To make the coloring allocator practical, the authors developed a method to trim the interference graph that significantly shrank the graph. The result was a state-of-the-art JIT that employed algorithms once thought to be too expensive for use in a JIT.

of loop iterations that it executes. Neither metric perfectly captures the notion that a method uses a large fraction of the running time. However, any method that does consume a significant fraction of runtime will almost certainly have a large value in one of those two metrics.

The system can "sum" the counters by using a single location for all the counters in a method.

Thus, the system should count both calls to a method and loop iterations within a method. Strategically placed profile counters can capture each of these conditions. For call counts, the system can insert a profile counter into each method's prolog code. For iteration counts, the system can insert a profile counter before each loop-closing branch. To trigger compilation, it can either use separate thresholds for loops and invocations, or it can sum the counters and use a single threshold.

HotSpot counted both calls and iterations and triggered a compilation when the combined count exceeded a preset threshold of 10,000 events. This threshold is much larger than the one used in Dynamo (50). It reflects the more aggressive and expensive compilation in HotSpot.

Runtime Profile Data

To capture profile data, compiler writers can instrument either the VM code for the application or the implementation of the VM-code engine.

Instrumented VM Code. The system can insert VM code into the method to increment and test the profile counters. In this design, the profile overhead executes as VM code. Either the AOT compiler or the Class Loader can insert the instrumentation. Counter for calls can be placed in the method's prolog code, while counters for a specific call-site can be placed in the appropriate precall sequence.

To profile loop iterations, the transformation can insert a counter into any block that is the target of a backward branch or jump. An AOT strategy might decrease the cost of instrumentation; for example, if the AOT compiler knows the number of iterations, it can increment the profile counter once for the entire loop.

Instrumented Engine. The compiler writer can embed the profile support directly into the implementtaion of the VM-code engine. In this scheme, the emulator's code for branches, jumps, and the call operation (e.g., the JVM's `invokestatic` or `invokevirtual`) can directly increment and test the appropriate counters, which are stored at preplanned locations. Because the profile code executes as native code, it should be faster than instrumenting the VM code.

The emulator could adopt the address comparison strategy of the trace optimizer to identify loop header blocks. If the target address is numerically smaller than the PC, the targeted block is a potential loop header. Alternatively, it could rely on the AOT compiler to provide an annotation that identifies loop-header blocks.

By contrast, an AOT compiler would find loop headers using dominators (see Section 9.2.1).

Compiling the Hot Method

When profile data triggers the JIT to compile some method x, the JIT can simply retrieve the VM code for x and compile it. The JIT resembles a full-fledged compiler. It parses the VM code into an IR, applies one or more passes of optimization to the IR, and generates native code—performing instruction selection, scheduling, and register allocation. The JIT writes that native code into the native-code cache (see Fig. 14.6). Finally, it updates the tables or pointers that the system uses to invoke methods so that subsequent calls map to the cached native code.

Tree-pattern matching techniques are a good match to the constraints of a JIT (see Section 11.4).

In a mixed-mode environment, the benefits of JIT compilation should be greater than they are in a native-code environment because the cold code executes more slowly. Thus, a hot-method optimizer in a mixed-mode environment can afford to spend more time per method on optimization and code generation. Hot-method optimizers have applied many of the classic scalar optimizations, such as value numbering, constant propagation, dead-code elimination, and code motion (see Chapters 8 and 10). Compiler writers choose specific techniques for the combination of compile-time efficiency and effectiveness at improving code.

GLOBAL VALUE NUMBERING

The literature on method-level JITs often mentions *global value numbering* as one of the key optimizations that these JITs employ. The dutiful student will find no consensus on the meaning of that term. Global value numbering has been used to refer to a variety of distinct and different algorithms.

One approach extends the ideas from local value numbering to a global scope, following the path taken in superlocal and dominator-based value numbering (DVNT). These algorithms are more expensive than DVNT and the cost-benefit tradeoff between DVNT and the global algorithm is not clear.

Another approach uses Hopcroft's partitioning algorithm to find operations that compute the same value, and then rewrites the code to reflect those facts. The HotSpot Server compiler used this idea, which fit well with its program dependence graph IR.

Finally, the JIT writer could work from the ideas in lazy code motion (LCM). This approach would combine code motion and redundancy elimination. Because LCM implementations solve multiple data-flow analysis problems, the JIT writer would need to pay close attention to the cost of analysis.

Optimizations

Hot-method JITs apply local, regional, and global optimizations. Because the JIT operates at runtime, the compiler writer can arrange for an optimization to access the runtime state of the executing program to obtain runtime values and use them in optimization.

Value Numbering Method-level JITs typically apply some form of value numbering. It might be a regional algorithm, such as DVNT, or it might be one of a number of distinct global algorithms.

Value numbering is attractive to the JIT writer because these techniques achieve multiple effects at a relatively low cost. Typically, these algorithms perform some subset of redundancy elimination, code motion, constant propagation, and algebraic simplification.

Inline method caches can provide site-specific data about receiver types. The idea can be extended to capture type information on parameters, as well.

Specialization to Runtime Data A JIT can have access to data about the code's behavior in the current execution, particularly values of variables, type information, and profile data. Runtime type information lets the JIT speculate; given that a value consistently had type t in the past, the JIT assumes that it will have type t in the future.

Such speculation can take the form of a fast-path/slow-path implementation. Fig. 14.7 shows, conceptually, how such a scheme might work. The code assumes that x and y are both 32-bit integers and tests for that case;

```
// $x \leftarrow y + z$
if $actual\_type(y) = 32\_bit\_integer$ and
  $actual\_type(z) = 32\_bit\_integer$ then
    $x \leftarrow 32\_bit\_integer\_add(y, z)$
else
    $x \leftarrow generic\_add(y, actual\_type(y), z, actual\_type(z))$;
```

■ **FIGURE 14.7** Code with a Fast Path for the Expected Case.

if the test fails, it invokes the generic addition routine. If the slow path executes too many times, the system might recompile the code with new speculated types (see the discussion of "deoptimization" on page 742).

Inline Substitution The JIT can inline calls, which lets it eliminate method lookup overhead and call overhead. Inlining leads the JIT to tailor the callee's body to the environment at the call site. The JIT can use data from inline method caches to specialize code based on call-site specific type data. It can also inline call sites in the callee; it should have access to profile data and method cache data for all of those calls.

The JIT can also look back to callers to assess the benefits of inlining a method into one or more of its callers. Again, profile data on the caller and type information from the inline cache may help the JIT make good decisions on when to inline.

When a JIT considers inline substitution, it has access to all of the code for the application. In an AOT compiler, that situation is unlikely.

Code Generation Instruction selection, scheduling, and register allocation can each further improve the JIT-compiled code. Effective instruction selection makes good use of the ISA's features, such as address modes. Instruction scheduling takes advantage of instruction-level parallelism and avoids hardware stalls and interlocks. Register allocation tries to minimize expensive spill operations.

The challenge for the JIT writer is to implement these passes efficiently. Tree-pattern matching techniques for selection combine locally optimal code choice with extreme JIT-time efficiency. Both the scheduler and the allocator can capitalize on sparsity to make global algorithms efficient enough for JIT implementation. The HotSpot Server Compiler demonstrated that efficient implementations can make these global techniques not only acceptable but advantageous.

As with any JAVA system, a JIT should also try to eliminate null-pointer checks, or move them to places where they execute less frequently. Escape analysis can discover objects whose lifetimes are guaranteed to be contained within the lifetime of some method. Such objects can be allocated in the method's AR rather than on the heap, with a corresponding decrease in allocation and collection costs. Final and static methods can be inlined.

Deoptimization

Deoptimization
the JIT generates less optimized code due to changes in runtime information

If the JIT uses runtime information to optimize the code, it runs the risk that changes in that data may render the compiled code either badly optimized or invalid. The JIT writer must plan for reasonable behavior in the event of a failed type speculation. The system may decide to *deoptimize* the code.

To "notice," the system would need to instrument the slow path.

Consider Fig. 14.7 again. If the system noticed, at some point, that most executions of this operator executed the *generic_add* path, it might recompile the code to speculate on another type, to speculate on multiple types, or to not speculate at all. If the change in behavior is due to a phase-shift in program behavior, reoptimization may help.

If, however, the statement has simply stopped showing consistency in the types of x, y, or z, then repeated reoptimization may be the wrong answer. Unless the compiled code executes enough to cover the cost of JIT compilation, the recompilations will slow down execution.

The alternative is to deoptimize the code. Depending on the precise situation and the depth of knowledge that the JIT has about temporal type locality, it might use one of several strategies.

- If the JIT knows that the actual type is one of a small number, it could generate fast path code for each of those types.
- If the JIT knows nothing except that the speculated type is often wrong, it might generate unoptimized native code that just calls *generic_add* or it might inline *generic_add* at the operation.
- If the JIT has been called too often on this code due to changing patterns, it might mark the code as not fit for JIT compilation, forcing the code to execute in emulation.

A deoptimization strategy allows the JIT to speculate, but limits the downside risk of incorrect speculation.

14.4.2 Hot-Methods in a Native-Code Environment

Several issues change when the JIT writer attempts hot-method optimization in a native-code environment. This section builds on insights from the Deutsch-Schiffman SMALLTALK-80 implementation.

Initial Compilations

The native-code environment must ensure that each method is compiled to native code before it runs. Many schemes will work, including a straightforward AOT compilation, load-time compilation of all methods, or JIT compilation on the first invocation, which we will refer to as *compile on* *call*. The first two options are easier to implement than the last one. Neither, however, creates the opportunity to use runtime information during that initial compilation.

A compile-on-call system will first generate code for the program's main routine. At each call site, it inserts a stub that (1) locates the VM code for the method; (2) invokes the JIT to produce native code for the method; and (3) relinks the call site to point to the newly compiled native code. When the execution first calls method *m*, it incurs a delay for the JIT to compile *m* and then executes the native-code version of *m*. If runtime facts are known, the first call to *m* can capitalize on them.

Using an indirect pointer to the code body (a pointer to a pointer) may simplify the implementation.

If the JIT compiler supports multiple levels of optimization, the compiler writer must choose which level to use in these initial compiles. A lower level of optimization should reduce the cost of the initial JIT compiles, at the cost of slower execution. A higher level of optimization might increase the cost of the initial JIT compiles, with the potential benefit of faster execution.

To manage this tradeoff, the system may use a low optimization level in the initial compiles and recompile methods with a higher level of optimization when they become hot. This approach, of course, requires data about execution frequencies and type locality.

Gathering Profile Data

In a native-code environment, the system can gather profile information in two ways: instrument the code to collect profile data at specific points in the code, or shift to interrupt-driven techniques that discover where the executable spends its time.

Instrumented Code. The JIT compiler can instrument code as described earlier for a mixed-mode hot-method optimizer. The JIT can insert code into method prologs to count total calls to the method. It can obtain call-site specific counts by adding code to precall sequences. It can insert code to count loop iterations before loop-closing branches.

With instrumented code, JIT invocation proceeds in the same way that it would in the mixed-mode environment. The JIT is invoked when execution counts, typically call counts and loop iterations, pass some preset threshold. For a loop-iteration count, the code to test the threshold and trigger compilation should be inserted, as well.

To capture opportunities for type speculation and type-based code specialization, the JIT can arrange to record the type or class of specific values—typically, parameters passed to the method or values involved at a call in the method. The JIT should have access to that information.

Interrupt-Driven Profiles. Method-invocation counts tell the system how often a method is called. Iteration counts tell the system how often a loop body executes. Neither metric provides insight into what fraction of total running time the instrumented code actually uses.

The system must produce tables to map an address into a specific location in the original code.

A native-code environment spends virtually all of its time executing application code. Thus, it can apply another strategy to discover hot code: interrupt-driven profiling. In this approach, the system periodically stops execution with a timer-driven interrupt. It maps the program-counter address at the time of the interrupt back to a specific method, and increments that method's counter. Comparing the method's counter against the total number of interrupts provides an approximation to the fraction of execution time spent in that method.

Some systems have used a combination of instrumented code and interrupt-driven profile data.

Because an interrupt-driven profile measures something subtly different than instrumented code measures, the JIT writer should expect that an interrupt-driven scheme will optimize different methods than an instrumented code scheme would.

The JIT still needs data, when possible, on runtime types to guide optimization. While some such data might exist in inline method caches at call sites, the system can only generate detailed information if the compiler adds type-profiling code to the executable.

Deoptimization with Native Code

When a compiled method begins execution, it must determine if the preconditions under which it was compiled (e.g., type or class speculation, constant valued parameters, etc.) still hold. The prolog code for a method can test any preconditions that the JIT assumed in its most recent compilation. In a mixed-mode environment, the system could execute the VM code if the precondition check fails; in a native-code environment, it must invoke the JIT to recompile the code in a way that allows execution to proceed.

In a recompilation, the system should attempt to provide efficient execution while avoiding situations where frequent recompilations negate the benefits of JIT compilation. Any one of several deoptimization strategies might make sense.

This strategy suggests a counter that limits the number of "phase shifts" the JIT will tolerate on a given method.

- The JIT could simply recompile the code with the current best runtime information. If the change in preconditions was caused by a phase shift in program behavior, the current preconditions might hold for some time.
- If the JIT supports multiple levels of optimization—especially with regard to type speculation—the system could instruct the JIT to use a lower level of speculation, which would produce more generic and less

tailored code. This approach tries to avoid the situation where the code for some method oscillates between two or more optimization states.

- An aggressive JIT could compile a new version of the code with the current preconditions and insert a stub to choose among the variant code bodies based on preconditions. This approach trades increased code size for the possibility of better performance.

The best strategy will depend on how aggressively the JIT uses runtime information to justify optimizations and on the economics of JIT compilation. If the JIT takes a small fraction of execution time, the JIT writer and the user may be more tolerant of repeated compilations. By contrast, if it takes multiple invocations of a method to compensate for the cost of JIT compilation, then repeated recompilations may be much less attractive than lowering the level of speculation and optimization.

The Economics of JIT Compilation

The fundamental tradeoff for the JIT writer is the difference between cycles spent in the JIT and cycles saved by the JIT. In a native-code environment, the marginal improvement from the JIT may be lower, simply because the unoptimized code runs more quickly than it would in a similar mixed-mode environment.

This observation, in turn, should drive some of the decisions about which optimizations to implement and how much recompilation to tolerate. The JIT writer must balance costs, benefits, and policies to create a system that, on balance, improves runtime performance.

SECTION REVIEW

A hot-method optimizer finds procedures that either execute frequently or occupy a significant fraction of execution time. It optimizes each procedure in light of the runtime facts that it can discern. Because a method optimizer can encounter control flow, it can benefit from regional and global optimizations, such as global value numbering or code motion; these transformations have higher costs and, potentially, higher payoffs than the local and superlocal techniques available to a trace optimizer.

The tradeoffs involved in a specific design depend on the execution environment, the source-language features that produce inefficiency, and the kinds of information gathered in the runtime environment. The design of a hot-method optimizer requires an understanding of the language, the system, the algorithms, and the behavior of the targeted applications.

REVIEW QUESTIONS

1. In a mixed-mode environment, what kinds of data could the emulator gather to facilitate specialization? How might it identify such opportunities and collect the necessary data?
2. Interrupt-driven profiles can work well in an environment where the dominant fraction of execution occurs in native code. What are some of the difficulties that the compiler writer might encounter using interrupt-driven profiles in a mixed-mode environment?

14.5 ADVANCED TOPICS

The previous sections introduce the major issues that arise in the design of a runtime optimizer. To build such a system, however, the compiler writer must make myriad design decisions, most of which have an impact on the effectiveness of the system. This section explores several major topics that arise in the literature that surrounds JIT compilation. Each of them has a practical impact on system design. Each of them can change the overall efficacy of the system.

14.5.1 Levels of Optimization

In practice, we know few developers who consider compile time when selecting an optimization level.

AOT compilers typically offer the end user a choice among multiple levels of optimization. This feature allows the user, in theory, to use stronger optimization in places where it matters, while saving on compile time in places where the additional optimization makes little difference.

A JIT-based system might provide multiple levels of optimization for several reasons.

- Because the elapsed time for application execution includes the JIT's execution, the JIT writer may decide to include in the standard compilation only those optimizations that routinely produce improvements.
- The system may find that a native-code fragment executes often enough to justify more extensive analysis and optimization, which requires more JIT time and saves more runtime.
- If the JIT performs speculation based on runtime information, such as types and classes, the JIT may later need to deoptimize the code, which suggests a lower level of optimization.

For all these reasons, some runtime optimization systems have implemented multiple levels of optimization.

If the system discovers a loop with a large iteration count, it might apply loop-specific optimizations, such as unrolling, strength reduction, or code motion. To ensure that those changes have immediate effect, it could perform on-stack replacement (see Section 14.5.2).

If the system finds that one method accounts for a significant fraction of interrupt-based profile points, it might apply deeper analysis and more intense optimization. For example, it might inline calls, perform analyses to disambiguate types and classes, and reoptimize.

In either of these scenarios, a JIT with multiple levels of optimization needs a clear set of policies to govern when and where it uses each level of optimization. One key part of that strategy will be a mechanism to prevent the JIT from trying to change the optimization level too often—driving up the JIT costs without executing the code enough to amortize the costs.

14.5.2 On-Stack Replacement

A method-level JIT can encounter a situation in which one of the profile counters crosses its threshold during an execution of a long-running method. For example, consider a method with a triply nested loop that has iteration counts of 100 at each level. With a threshold of 10,000, the counter on the inner loop would trigger compilation after just one percent of the iterations.

The counter shows that the method is hot and should be optimized. If the system waits until the next call to "install" the optimized code, it will run the current code for the rest the current invocation. In the triply nested loop, the code would run 99 percent of the iterations after the counter had crossed the threshold for optimization.

In effect, the system behaves as if a long-running method has a higher threshold to trigger compilation.

To avoid this missed opportunity, the system could pause the execution, optimize and compile the code, and resume the execution with the improved code. This approach capitalizes on the speed of the compiled code for the majority of the loop iterations. To resume execution with the newly optimized code, however, the system must map the runtime state of the paused execution into the runtime state needed by the newly optimized code.

This approach, optimizing the procedure in a way that the current invocation can continue to execute, is often called *on-stack code replacement*. The JIT builds code that can, to the extent possible, execute in the current runtime environment. When it cannot preserve the values, it must arrange to map values from the current environment into the new environment.

On-stack code replacement
A technique where the runtime system pauses execution, JIT compiles the executing procedure, and resumes execution with the newly compiled code

The JIT can use its detailed knowledge of the old code to create the new environment. It can generate a small stub to transform the current environment—values in registers plus the current activation record—into the environment need by the new code.

- The stub may need to move some values. The storage map of the new code may not match the storage map of the old code.
- The stub may need to compute some values. Optimizations such as code motion or operator strength reduction may create new values that did not exist in the original code.
- The stub may be able to discard some values. The state of the original code may contain values that are dead or unused in the new code.

The stub runs once, before the first execution of the new code. At that point, it can be discarded. If the JIT runs in a separate thread, as many do, the system needs some handshaking between the JIT and the running code to determine when it should switch to the new code.

The compiler writer has several ways to reduce the complexity of on-stack replacement.

- She can limit the number of points in the code where the system can perform replacement. The start of a loop iteration is a natural location to consider. Execution of the next iteration begins after compilation and state mapping.
- She can simplify the state-mapping problem by limiting the set of optimizations that the JIT uses when compiling for on-stack replacement. In particular, the JIT might avoid techniques that require significant work to map the old environment into the new one.

Techniques that create compensation code or introduce new values can complicate the mapping. Examples include code motion, software pipelining, and inline substitution.

The implementation of on-stack replacement ties in a fundamental way to the interfaces between emulated and compiled codes and their runtime environments. The details will vary from system to system. This strategy has the potential to provide significant improvement in the performance of long-running methods.

14.5.3 **Code Cache Management**

To avoid confusion, we will refer to the JIT's cache as a *code cache* and to processor caches as *hardware caches*.

Almost all JIT-based systems build and maintain a code cache—a dedicated, software-managed block of memory that holds JIT-compiled code. The JIT writer must design policies and build mechanisms to manage the code cache. Conceptually, code caches blend the problems and policies of a hardware cache and a software-managed heap.

- Hardware caches determine an object's placement by an arithmetic mapping of virtual addresses to physical addresses. In a heap, software searches for a block of free space that will hold the object. Code caches are closer to the heap model for placement.

- Hardware caches deal with fixed sized blocks. Heaps deal with requests for arbitrarily sized blocks, but often round those requests to some common sizes. Code caches must accommodate blocks of native code of different sizes.
- Hardware caches use automatic, policy-based eviction schemes, typically informed by the pattern of prior use. Heaps typically run a collection phase to find blocks that are no longer live (see Section 6.6.2). Code caches use policies and mechanisms similar to hardware caches.

Most JIT-based systems have a separate code cache for each process or each thread. Some JIT writers have experimented with a global code cache, to allow the reuse of JIT compiled code across processes. The primary benefit from these designs appears to be a reduction in overall memory use; they may provide better performance for a multitasked environment on a limited memory system. When these systems find cross-process sharing, they also avoid reinvoking the JIT on previously compiled code, which can reduce overall runtimes.

The use of a limited-size code cache suggests that the standard virtual-memory paging mechanism is either too slow or too coarse-grained to provide efficient support for the JIT-compiled code. Use of a limited-size cache also implies that a code-cache eviction will discard the JIT-compiled code; delinking it from the executing program and necessitating either emulation or a recompilation if it is invoked in the future.

If virtual memory is fast enough, the system can make the cache large and let the paging algorithms manage the problem.

Replacement Algorithm

When the JIT compiles code, it must write that code into the code cache. If the cache management software cannot find a block of unused memory large enough to hold the code, it must evict another segment from the cache.

Replacement in the code cache differs from replacement in a hardware cache. A set-associative hardware cache determines the set to which the new block maps and evicts one of the set's blocks. The literature suggests evicting the least recently used (LRU) block; many hardware caches use or approximate LRU replacement.

A direct-mapped hardware cache has a set size of one.

Code cache management algorithms need to evict enough segments to create room for the newly compiled code. In a hardware cache, eviction involves a single fixed-size line. In a software-managed code cache, allocation occurs at the granularity of the segment of compiled code (a trace, a method, or multiple methods). This complicates both the policy and the implementation of the replacement algorithm.

The cache management software should evict from the code cache one or more segments that have not been used recently. The evicted segments must free enough space to accommodate the new code without wasting "too much" space. Choosing the LRU segment might be a good start, but the potential need to evict multiple segments complicates that decision. If the new code requires eviction of multiple segments, those segments must be adjacent. Thus, implementing an LRU mechanism requires some additional work.

The final constraint on replacement is that the algorithms must be fast; any time spent in the replacement algorithms adds to the application's running time. Creative engineering is needed to minimize the cost of choosing a block to evict and of maintaining the data structures to support that decision.

Fragmentation

Repeated allocation and replacement can fragment the space in a code cache. Collected heaps address fragmentation with compaction; uncollected heaps try to merge adjacent free blocks. Code caches lack the notion of a `free` command; in general, it is unknowable whether some code fragment will execute in the future, or when it will execute.

If the system executes in a virtual-memory environment, it can avoid some of the complication of managing fragmentation by using more virtual address space than allocated memory. As long as the code cache's working set remains within the intended cache size, the penalty for using more virtual address space should be minimal.

14.5.4 Managing Changes to the Source Code

This feature is not new. Both APL in the 1960s and Smalltalk in the 1970s had features to edit source code. Those systems, however, were built on interpreters.

Some languages and systems allow runtime changes to an application at the source-code level. Interpreted environments handle these changes relatively easily. If the runtime environment includes JIT-compiled code, the system needs a mechanism to recognize when a change invalidates one or more native-code fragments, and to replace or recompile them.

The runtime system needs three mechanisms. It must recognize when change occurs. It must identify the code and data that the change affects. Finally, it must bring the current runtime state into agreement with the new source code.

Recognize Change

The system must know when the underlying code has changed. The most common way to capture changes is by restricting the mechanisms for making a change. For example, JAVA code changes require the class loader; in APL, code changes involved use of the *quote-quad* operator. The interface that allows the change can alert the system.

Identify Scope of Change

The system must understand where the changes occur. If the text of a procedure fee changes, then native code for fee is undoubtedly invalid. The system needs a map from a procedure name to its native-code implementation. The more subtle issues arise when a change in fee affects other procedures or methods.

Similar problems arise with interprocedural optimization in an AOT compiler (see Section 8.7.3).

If, for example, the JIT previously inlined fee into its caller foe, then the change to fee also invalidates the prior compilation of foe. If fee is a good target for inline substitution—say, its code size is smaller than the standard linkage code—then a change to fee might trigger a cascade of recompilations. The map from procedure names to code bodies becomes multivalued.

Interface changes to a method, such as changing the parameters, must invalidate both the changed procedure and all of the procedures that call it. Details in the precall and postreturn sequences are inferred from the interface; if it changes, those sequences likely change, too.

Recompiling Changed Code

At a minimum, the system must ensure that future calls to a method execute the most recent code. In a mixed-mode environment, it may suffice to delete the JIT-compiled code for a changed method and revert to interpreting the VM code. When the method becomes hot again, the system will compile it. In a native-code environment, the system must arrange for the new code to be compiled—either aggressively or at its next call.

To simplify recompilation, the JIT writer might add a level of indirection to each call. The precall sequence then refers to a fixed location for the callee; the JIT stores a pointer to the code body at that location. The extra indirection avoids the need to find all of the callers and update their code pointers. To relink the method, the JIT simply overwrites the one code pointer.

In the case where the changed code invalidates compilations of other procedures, the number of invalidations can rise, but the same basic mechanisms should work.

Changes to Declarations

Runtime changes to the source code introduce a related problem—one that arises in both interpreted and compiled implementations. If the source code can change, then the definitions of data objects can change. Consider, for example, a change that adds a new data member to a class. If that class already has instantiated objects, those objects will lack the new data member. The source language must define how to handle this situation, but in the worst case, the system might need to find and reformat all of the instantiated instances of the class—a potentially expensive proposition.

To simplify finding all of the objects in a class, the system might link them together. Early SMALLTALK systems exhaustively searched memory to find such objects; the limited memory on those systems made that approach feasible.

14.6 SUMMARY AND PERSPECTIVE

Just-in-time compilation systems make optimization and code generation decisions at runtime. This approach can provide the JIT compiler with access to more precise information about names, values, and bindings. That knowledge, in turn, can help the JIT specialize the code to the actual runtime environment.

JIT systems operate under strict time constraints. Well-designed and well-built systems can provide consistent improvements. The speedups from the JIT must compensate for the time spent gathering information, making decisions, and compiling code. Thus, JIT writers need a broad perspective on language implementation and a deep knowledge of compilation and optimization techniques.

Despite the long history of runtime optimization, the field remains in flux. For example, one of the most heavily used JITs, Google's V8 JAVASCRIPT JIT, was originally written as a native-code, compile-on-call system. Experience led to a reimplementation that uses a mixed-mode, hot-method approach. The primary justification for this change given in the literature was to reduce code space and startup time; the hot-method version also avoided parsing unused code. Changes in languages, runtime environments, and experience have driven work in runtime optimization over the last decade. We anticipate that this field will continue to change for years to come.

CHAPTER NOTES

Runtime compilation has a long history. McCarthy included a runtime compilation facility in his early LISP system so that it could provide efficient execution of code that was constructed at runtime—a direct consequence of LISP's unified approach to code and data [266].

Thompson used an "edit-time" compilation of regular expressions into naive code for the IBM 7094 to create a powerful textual search tool for his port of the QED editor [345]; Section 2.4.2 describes the underlying construction.

Hansen built his Adaptive Fortran system to explore the practicality and profitability of runtime optimization [188]. It supported a subset of FORTRAN IV and a small set of optimizations. He modeled the behavior of his system against widely known FORTRAN compilers of the time. His dissertation includes significant discussion on how to estimate the benefits of an optimization and how to trigger the runtime optimizations.

The Deutsch-Schiffman Smalltalk-80 implementation, built for an early Sun Microsystems workstation, demonstrated the potential of runtime compilation for improving dynamic languages [137]; contemporary implementations that relied on interpreting Smalltalk bytecode ran more slowly.

The HotSpot Server Compiler [288] and Dynamo [32] were widely recognized and influential systems. HotSpot influenced design decisions in JIT-based systems for a decade or more. Dynamo inspired a generation of work on problems that ranged from code-cache management to software dynamic translation.

Most method-level optimizers apply some form of global value numbering. These algorithms range from global extensions of the ideas in local value numbering [59,167] through algorithms that build on Hopcroft's DFA minimization algorithm [90,312] to implementations of lazy code motion (see the notes for Chapter 10).

The time constraints that arise in JIT compilation have encouraged the use of efficient algorithms. Tree-pattern matching instruction selectors can be hyper-efficient: using five to ten compile-time operations per emitted operation [162,163,297]. Linear scan register allocation avoids the expensive part of a full-fledged coloring allocator: building the interference graph [296]. In an environment where many methods are small and do not require spill code, linear scan works well. The HotSpot Server Compiler used interference graph trimming to reduce the cost of a full-fledged coloring allocator [93].

EXERCISES

Section 14.2

1. Consider again the plot in Fig. 14.1 (JAVA scaling with and without the JIT). How might changes in the threshold for JIT compilation affect the behavior of the JIT-enabled curve, particularly at the lower end of the curve, shown in panel (b)?

Section 14.3

2. Write pseudocode for a backward pass over an acyclic trace that discovers dead and partially dead operations. Assume that the JIT has LIVE information at each exit from the trace.

   How might the JIT obtain LIVE information for the trace exits?

3. One consideration in the design of a hot-trace optimizer is how to handle intermediate entries into a trace. The design in Section 14.3 ignores intermediate entries, with the effect that multiple copies of some blocks are made.

   As an alternative, the compiler writer could have the trace-building algorithm split the trace at an intermediate entry. This strategy would generate an optimized trace for the portion before the intermediate entry and an optimized trace for the portion after the intermediate entry. It would then directly link the former part to the latter part.

   a. How might the trace-building algorithm recognize that a block is an intermediate entry point?
   b. What kinds of JIT optimizations might have reduced effectiveness as a result of splitting traces around intermediate entry points?

4. If the trace optimizer has a bounded code cache and it fills that cache, it may need to evict one or more traces.

   a. What complications do linked traces introduce?
   b. What approaches can the code-cache management algorithms take to managing the eviction of linked traces?

Section 14.4

5. When a system with a hot-method optimizer discovers that some method has triggered too many recompilations, it may decide to deoptimize the method.

   The JIT could treat individual call sites differently, linking each call site to an appropriately optimized code body for the method.

   Assume, for the moment, that code-cache space is not an issue.

   a. What information should the system gather to enable such call-site specific optimization and deoptimization?
   b. What additional runtime data structures might the JIT need in order to implement such call-site specific optimization?
   c. One obvious cost of such a scheme is space in the code cache. How might the compiler writer limit the proliferation of variant code bodies for a single method?

6. Some hot-method JITs compile code in a separate thread, asynchronously. What advantages might this offer to the end user? What disadvantages might it create?

7. Deoptimization must deal with the results of inline substitution. Suppose the JIT has inlined `fie` into `fee`, and that it later decides that it must deoptimize `fee`. What strategies can the JIT implement to simplify deoptimization of a method that includes inlined code?

8. Ahead-of-time (AOT) compilers choose which procedures to inline based on some combination of static analysis and profile data from a prior run on “representative” data. By contrast, a JIT decides to inline a procedure based almost entirely on runtime profile information.

   a. Suggest three heuristics that a hot-method JIT might use to determine whether or not to inline the callee at a specific call site.

   b. What kinds of data can the runtime system gather to help in the decision of whether or not to inline a specific call site?

   c. Contrast your strategies, and the results you expect from them, with the results you expect from an AOT compiler.

This page intentionally left blank

Appendix A

ILOC

A.1 INTRODUCTION

ILOC is a linear assembly code for a simple abstract RISC machine. The ILOC used in this book is a simplified version of the intermediate representation (IR) that was used in the Massively Scalar Compiler Project (MSCP) at Rice University. For example, ILOC as defined here assumes just two base data types, an integer of unspecified length, and a single character. In the MSCP compiler, the IR supported a much broader set of data types.

The ILOC abstract machine has an unlimited number of registers. It has register-to-register operations; load and store operations; comparisons; and branches. It supports four memory address modes: *direct*, *address-immediate*, *address-offset*, and *immediate*. Source operands are read at the start of the cycle in which the operation issues. Result operands are defined at the end of the cycle in which the operation completes.

Other than its instruction set, the details of the machine are left unspecified. Most of the examples in this book assume a simple machine, with a single functional unit that executes ILOC operations in their order of appearance. When other configurations are used, we discuss them explicitly.

An ILOC program consists of a sequential list of instructions, as follows:

$$
\begin{array}{lcl}
\textit{IlocProgram} & \rightarrow & \textit{InstructionList} \\
\textit{InstructionList} & \rightarrow & \textit{Instruction} \\
 & | & \textit{Instruction InstructionList}
\end{array}
$$

An instruction consists of one or more operations, with an optional label.

$$
\begin{array}{lcl}
\textit{Instruction} & \rightarrow & \textit{OperationList} \\
 & | & \texttt{label}\ \underline{:}\ \textit{OperationList}
\end{array}
$$

A label is an alphanumeric string, $([A\text{–}Z] \mid [a\text{–}z])\,([A\text{–}Z] \mid [a\text{–}z] \mid [0\text{–}9])^*$. A colon follows the label. If the code needs multiple labels at some point, we insert labeled `nop` instructions at the appropriate location.

An *OperationList* consists of either a single *Operation*, or a list of *Operation*s surrounded by square brackets and separated by semicolons.

$$
\begin{array}{lcl}
\mathit{OperationList} & \rightarrow & \mathit{Operation} \\
 & | & [\ \mathit{ListOfOps}\ ] \\
\mathit{ListOfOps} & \rightarrow & \mathit{Operation} \\
 & | & \mathit{Operation}\ \ ;\ \mathit{ListOfOps} \\
\mathit{Operation} & \rightarrow & \mathit{NormalOp} \\
 & | & \mathit{ControlFlowOp}
\end{array}
$$

An ILOC operation corresponds to a single machine-level operation. An *Operation* is either a *NormalOp* that performs computation or data movement, or it is a *ControlFlowOp* used to change the flow of control in the program.

*NormalOp*s consist of an opcode, a list of comma-separated source operands, and a list of comma-separated result operands. The sources are separated from the results by the symbol ⇒, pronounced "into."

$$
\begin{array}{lcl}
\mathit{NormalOp} & \rightarrow & \mathit{Opcode}\ \mathit{OperandList} \Rightarrow \mathit{OperandList} \\
\mathit{OperandList} & \rightarrow & \mathit{Operand} \\
 & | & \mathit{Operand}\ \underline{,}\ \mathit{OperandList} \\
\mathit{Operand} & \rightarrow & \texttt{register} \\
 & | & \texttt{number} \\
 & | & \texttt{label}
\end{array}
$$

Unfortunately, as in a real assembly language, the relationship between an opcode and the form of its operands is not systematic. The easiest way to specify the form of the operands for each opcode is in a tabular form. The tables in Section A.6 show the full set of opcodes and their operands for each of the ILOC operations that appear in the book.

*Operand*s may be one of three types: `register`, `number`, or `label`. The type of each operand is determined by the opcode and the position of the operand in the operation. In the examples, we use both numerical (r_{10}) and symbolic (r_i) names for registers. Numbers are simple positive integers. Labels are, as defined earlier, alphanumeric strings, $(\,[A\text{–}Z] \mid [a\text{–}z]\,)\ (\,[A\text{–}Z] \mid [a\text{–}z] \mid [0\text{–}9]\,)^*$.

While the definition of a label allows `r10` and `ri` as labels, we avoid labels that could have other interpretations.

Arithmetic operations have one result; some of the `store` operations have two results, as do all of the branches.

As an example, `storeAI`, the *address-immediate* store operation, has one source operand (a `register`) and two result operands (a `register` and a `number`). The operation `storeAI` $r_i \Rightarrow r_j, 4$ adds 4 to the value in r_j to form an address and stores the value r_i into the memory location at that address. It performs the action: $\textsf{MEMORY}\,(r_j + 4) \leftarrow \textsf{CONTENTS}(r_i)$.

A.2 NAMING CONVENTIONS

The ILOC code used in examples throughout the text follows a simple set of naming conventions.

1. Memory offsets for variables are represented symbolically by prefixing the variable name with the `@` character.
2. The user can assume an unlimited supply of registers. These are named with simple integers, as in r_{1776}, or with symbolic names, as in r_i.
3. The register r_{arp} is reserved for a pointer to the current activation record. Thus, the operation `loadAI` r_{arp}, @x ⇒ r_1 loads an integer from offset `@x` in the current activation record into r_1.

ILOC comments begin with the string // and continue until the end of a line. We assume that these are stripped out by the scanner.

A.3 COMPUTATIONAL OPERATIONS

The first major category of operations in ILOC is the set of basic computational operations. ILOC provides these operations in both a three-address, register-to-register format and a two-register format that has an immediate constant as its second source operand.

Register-to-Register Operations

Opcode	Sources	Results	Meaning
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
mult	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
div	r_1, r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
lshift	r_1, r_2	r_3	$r_1 \ll r_2 \Rightarrow r_3$
rshift	r_1, r_2	r_3	$r_1 \gg r_2 \Rightarrow r_3$

The first four ocodes implement standard arithmetic operations. The latter two opcodes implement logical shift operations.

A real ILOC processor or a full-fledged compiler would need more than one arithmetic data type. This would lead to typed opcodes or to polymorphic opcodes. The MSCP compiler had distinct arithmetic operations for various lengths of integer and floating-point numbers, as well as for pointers.

The next group of opcodes specify register-immediate operations. Each of them takes as input a register and an immediate constant value. The noncommutative operations have two distinct forms to allow the constant as either the first or second value in the operation. For example, rsubI subtracts the value in r_1 from the immediate constant c_2.

Register-Immediate Operations			
Opcode	Sources	Results	Meaning
addI	r_1, c_2	r_3	$r_1 + c_2 \Rightarrow r_3$
subI	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$
rsubI	r_1, c_2	r_3	$c_2 - r_1 \Rightarrow r_3$
multI	r_1, c_2	r_3	$r_1 \times c_2 \Rightarrow r_3$
divI	r_1, c_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
rdivI	r_1, c_2	r_3	$c_2 \div r_1 \Rightarrow r_3$
lshiftI	r_1, c_2	r_3	$r_1 \ll c_2 \Rightarrow r_3$
rshiftI	r_1, c_2	r_3	$r_1 \gg c_2 \Rightarrow r_3$

A register-immediate operation is useful for three distinct reasons.

- It lets the compiler or the assembly-level programmer use a small constant directly in the operation. Thus, it decreases demand for registers. It may also eliminate an operation to load the constant into a register.
- It "reads" the constant from the instruction stream and, thus, from the instruction cache. Most processors have separate caches and data paths for instructions and for data; thus, the immediate operation uses fewer resources on the data-side of the memory interfaces.
- The immediate form of the operation records the value of the constant in a visible and easily accessible way, which ensures that optimization and code generation can see and use the value.

A.4 DATA MOVEMENT OPERATIONS

The second major category of operations in ILOC allow the compiler or the assembly-level programmer to specify data movement. We include two specialized operations in this section: a conditional move operation and a representation for a ϕ-function (see Section 4.6.2).

Memory Operations

ILOC provides a set of operations to move values between memory and registers: load and store operations. The examples in the book limit themselves to integer and character data; a complete version of ILOC would need load and store operations for a much broader variety of base types.

To produce efficient code, a compiler's back end must make effective use of the address modes provided on `load` and `store` operations. ILOC provides a variety of address modes in both the `load` and `store` operations.

Most processors use a separate adder to perform the arithmetic in an address computation. Thus, moving an addition into the address mode frees up an issue slot on another functional unit.

Load Operations

Opcode	Sources	Results	Meaning
`load`	r_1	r_2	MEMORY(r_1) ⇒ r_2
`loadAI`	r_1, c_2	r_3	MEMORY($r_1 + c_2$) ⇒ r_3
`loadAO`	r_1, r_2	r_3	MEMORY($r_1 + r_2$) ⇒ r_3
`cload`	r_1	r_2	character `load`
`cloadAI`	r_1, c_2	r_3	character `loadAI`
`cloadAO`	r_1, r_2	r_3	character `loadAO`
`loadI`	c_1	r_2	c_1 ⇒ r_2

The load operations differ in the address modes that they support.

- The *direct loads*, `load` and `cload`, use the value in r_1 as an address.
- The *address-immediate loads*, `loadAI` and `cloadAI`, add the immediate constant and the value in the source register to form the address.
- The *address-offset loads*, `loadAO` and `cloadAO`, add the contents of the two source registers to form the address.
- The *immediate load*, `loadI`, moves the constant into the target register.

A complete, ILOC-like IR with multiple base types for values should have an immediate load for each distinct base type.

The store operations support the same address modes as the load operations.

Store Operations

Opcode	Sources	Results	Meaning
`store`	r_1	r_2	r_1 ⇒ MEMORY(r_2)
`storeAI`	r_1	r_2, c_3	r_1 ⇒ MEMORY($r_2 + c_3$)
`storeAO`	r_1	r_2, r_3	r_1 ⇒ MEMORY($r_2 + r_3$)
`cstore`	r_1	r_2	character `store`
`cstoreAI`	r_1	r_2, c_3	character `storeAI`
`cstoreAO`	r_1	r_2, r_3	character `storeAO`

ILOC has no store immediate operation.

Register-to-Register Copy Operations

To move values directly between registers ILOC includes a set of register-to-register copy operations.

Copy Operations			
Opcode	Sources	Results	Meaning
`i2i`	r_1	r_2	$r_1 \Rightarrow r_2$ for integer values
`c2c`	r_1	r_2	$r_1 \Rightarrow r_2$ for character values
`c2i`	r_1	r_2	convert character to integer
`i2c`	r_1	r_2	convert integer to character

`i2i` and `c2c` simply copy a value from one register to another. By contrast, `c2i` and `i2c` convert the value to another type as part of the copy operation.

Conditional Move Operation

Many ISAs provide a conditional move operation that selects a value from one of two source registers based on a Boolean value in a third register. To enable examples that discuss this kind of operation, ILOC includes four-operand conditional move operations for integers and characters.

Architects include conditional move operations in an ISA to let the compiler avoid branches on particularly simple conditional constructs.

Conditional Move Operations			
Opcode	Sources	Results	Meaning
`c_i2i`	r_b, r_1, r_2	r_3	$r_1 \Rightarrow r_3$, *if* r_b = true $r_2 \Rightarrow r_3$, *otherwise*
`c_c2c`	r_b, r_1, r_2	r_3	$r_1 \Rightarrow r_3$, *if* r_b = true $r_2 \Rightarrow r_3$, *otherwise*

Representing ϕ Functions

When a compiler builds SSA form, it must represent ϕ-functions. In ILOC, the natural way to write a ϕ-function is as an ILOC operation:

$$\texttt{phi}\ r_i,\ r_j,\ r_k \Rightarrow r_m$$

for the ϕ-function $r_m \leftarrow \phi\,(r_i, r_j, r_k)$. Because of the nature of SSA form, the `phi` operation may take an arbitrary number of sources. It always defines a single target register. Representing ϕ-functions in this way poses some interesting engineering issues in the implementation of the IR.

A.5 CONTROL-FLOW OPERATIONS

Control-flow operations allow the ILOC program to sequence the execution of labeled blocks of operations. ILOC includes jumps, comparisons, and conditional branches. To facilitate the discussion in Section 7.4.2, ILOC also includes a syntax to represent predicated operations.

While control-flow operations still fit the three-address form of most ILOC operations, they vary in the meanings of their various operands. As a visual cue, the operations that involve a transfer of control (or an assignment to the program counter) use a single arrow, →, to separate sources from targets.

$$
\begin{array}{lclcl}
\mathit{ControlFlowOp} & \rightarrow & \texttt{jumpI} & & \rightarrow \texttt{label} \\
 & | & \texttt{jump} & & \rightarrow \texttt{register} \\
 & | & \mathit{Branch} & \texttt{register} & \rightarrow \texttt{label}\,\underline{,}\,\texttt{label} \\
 & | & \mathit{Compare} & \texttt{register}, \texttt{register} & \Rightarrow \texttt{register}
\end{array}
$$

ILOC provides two styles of *Branch* and *Compare* operations so that Section 7.4.2 can discuss translation of control-flow constructs for each of them.

Jumps

ILOC has both *jump immediate* and *jump to register* operations. Most examples use an immediate jump because it exposes the destination label.

Opcode	Sources	Results	Meaning
`jumpI`	--	$\texttt{l}_1$	$\texttt{l}_1 \rightarrow \texttt{PC}$
`jump`	--	$\texttt{r}_1$	$\texttt{r}_1 \rightarrow \texttt{PC}$

The `jump` operation is ambiguous because the target address is in a register. The compiler may be unable to deduce the correct set of possible target labels for a `jump`. This effect complicates CFG construction and can degrade the quality of information that data-flow analysis can derive. For these reasons, we prefer `jumpI` over `jump` whenever possible and practical.

Sometimes the compiler must generate a `jump` rather than a `jumpI`. For these situations, ILOC includes a pseudooperation that lets the compiler record the set of possible labels for a `jump` operation. The `tbl` pseudooperation has two arguments, a register and an immediate label.

Opcode	Sources	Results	Meaning
tbl	r_1, l_2	--	r_1 *might hold* l_2

A tbl operation can occur only after a jump. The sequence

```
jump          → ri
tbl   ri, L01
tbl   ri, L03
tbl   ri, L05
tbl   ri, L08
```

asserts that the jump targets one of L01, L03, L05, or L08, and no other label.

Comparisons and Conditional Branches

All branches in ILOC have a label for the true path and the false path. ILOC does not have the notion of a "fall-through" path.

ILOC supports two models for branches. The standard ILOC model, shown in Fig. A.1(a), uses a simple conditional branch, cbr, and a set of six Boolean-valued comparison operations. cbr tests the Boolean value produced by a comparison and branches to the appropriate immediate label.

The second branch model, shown in panel (b), has one comparison operation and a set of six conditional branches. It models the situation on an ISA where comparison sets a "condition code" (see Section 7.4.2). This model pushes the choice of relational operator into the branch operation.

Actual ISAs vary in the number of condition code registers they support.

In the examples, we designate the target of comp as a condition-code register by writing it as cc_i. The comp operation writes a value drawn from the set {LT, LE, EQ, GE, GT, NE,} into cc_i. The conditional branch has a variant for each of these six values.

Predicated Execution

Predication is another feature introduced to let the compiler or assembly programmer avoid a conditional branch. A predicated operation takes a Boolean value that determines whether or not the operation's result takes effect. In ILOC, a predicated operation is written as follows:

```
(rp) ? add r1, r2  ⇒  r3
```

This notation indicates that r_3 receives the sum of the values of r_1 and r_2 if r_p contains the value true. If r_p does not contain true, then r_3 is unchanged.

In principle, any ILOC operation can be predicated. In practice, predication does not make sense for some operations, such as a phi or a jump. (A predicated jump is just a cbr whose false path is the next operation.)

Opcode	Sources	Results	Meaning	
cmp_LT	r_1, r_2	r_3	true $\Rightarrow r_3$	*if* $r_1 < r_2$
			false $\Rightarrow r_3$	*otherwise*
cmp_LE	r_1, r_2	r_3	true $\Rightarrow r_3$	*if* $r_1 \leq r_2$
			false $\Rightarrow r_3$	*otherwise*
cmp_EQ	r_1, r_2	r_3	true $\Rightarrow r_3$	*if* $r_1 = r_2$
			false $\Rightarrow r_3$	*otherwise*
cmp_GE	r_1, r_2	r_3	true $\Rightarrow r_3$	*if* $r_1 \geq r_2$
			false $\Rightarrow r_3$	*otherwise*
cmp_GT	r_1, r_2	r_3	true $\Rightarrow r_3$	*if* $r_1 > r_2$
			false $\Rightarrow r_3$	*otherwise*
cmp_NE	r_1, r_2	r_3	true $\Rightarrow r_3$	*if* $r_1 \neq r_2$
			false $\Rightarrow r_3$	*otherwise*
cbr	r_1	l_2, l_3	$l_2 \rightarrow$ PC	*if* $r_1 =$ true
			$l_3 \rightarrow$ PC	*otherwise*

(a) The Standard Branch Syntax

Opcode	Sources	Results	Meaning	
comp	r_1, r_2	cc_3	*sets* cc_3	
cbr_LT	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	*if* $cc_3 =$ LT
			$l_3 \rightarrow$ PC	*otherwise*
cbr_LE	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	*if* $cc_3 =$ LE
			$l_3 \rightarrow$ PC	*otherwise*
cbr_EQ	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	*if* $cc_3 =$ EQ
			$l_3 \rightarrow$ PC	*otherwise*
cbr_GE	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	*if* $cc_3 =$ GE
			$l_3 \rightarrow$ PC	*otherwise*
cbr_GT	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	*if* $cc_3 =$ GT
			$l_3 \rightarrow$ PC	*otherwise*
cbr_NE	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	*if* $cc_3 =$ NE
			$l_3 \rightarrow$ PC	*otherwise*

(b) Branch Syntax to Model an ISA with Condition Codes

■ **FIGURE A.1** Conditional Branch Syntax.

A.6 OPCODE SUMMARY TABLES

The following pages contain summary tables that list all of the ILOC operations and their meanings. They are provided for reference.

Arithmetic Operations

Opcode	Sources	Results	Meaning
nop	*none*	*none*	*single cycle, side-effect free operation*
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
mult	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
div	r_1, r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
addI	r_1, c_2	r_3	$r_1 + c_2 \Rightarrow r_3$
subI	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$
rsubI	r_1, c_2	r_3	$c_2 - r_1 \Rightarrow r_3$
multI	r_1, c_2	r_3	$r_1 \times c_2 \Rightarrow r_3$
divI	r_1, c_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
rdivI	r_1, c_2	r_3	$c_2 \div r_1 \Rightarrow r_3$
lshift	r_1, r_2	r_3	$r_1 \ll r_2 \Rightarrow r_3$
lshiftI	r_1, c_2	r_3	$r_1 \ll c_2 \Rightarrow r_3$
rshift	r_1, r_2	r_3	$r_1 \gg r_2 \Rightarrow r_3$
rshiftI	r_1, c_2	r_3	$r_1 \gg c_2 \Rightarrow r_3$
and	r_1, r_2	r_3	$r_1 \wedge r_2 \Rightarrow r_3$
andI	r_1, c_2	r_3	$r_1 \wedge c_2 \Rightarrow r_3$
or	r_1, r_2	r_3	$r_1 \vee r_2 \Rightarrow r_3$
orI	r_1, c_2	r_3	$r_1 \vee c_2 \Rightarrow r_3$
xor	r_1, r_2	r_3	$r_1 \mathit{xor}\ r_2 \Rightarrow r_3$
xorI	r_1, c_2	r_3	$r_1 \mathit{xor}\ c_2 \Rightarrow r_3$

Data Movement Operations

Opcode	Sources	Results	Meaning
loadI	c_1	r_2	$c_1 \Rightarrow r_2$
load	r_1	r_2	$\text{MEMORY}(r_1) \Rightarrow r_2$
loadAI	r_1, c_2	r_3	$\text{MEMORY}(r_1 + c_2) \Rightarrow r_3$
loadAO	r_1, r_2	r_3	$\text{MEMORY}(r_1 + r_2) \Rightarrow r_3$
cload	r_1	r_2	*character* load
cloadAI	r_1, c_2	r_3	*character* loadAI
cloadAO	r_1, r_2	r_3	*character* loadAO
store	r_1	r_2	$r_1 \Rightarrow \text{MEMORY}(r_2)$
storeAI	r_1	r_2, c_3	$r_1 \Rightarrow \text{MEMORY}(r_2 + c_3)$
storeAO	r_1	r_2, r_3	$r_1 \Rightarrow \text{MEMORY}(r_2 + r_3)$
cstore	r_1	r_2	*character* store
cstoreAI	r_1	r_2, c_3	*character* storeAI
cstoreAO	r_1	r_2, r_3	*character* storeAO
i2i	r_1	r_2	$r_1 \Rightarrow r_2$ *for integers*
c2c	r_1	r_2	$r_1 \Rightarrow r_2$ *for characters*
c2i	r_1	r_2	*convert character to integer*
i2c	r_1	r_2	*convert integer to character*
phi	r_1, r_2, r_3	r_4	$\phi(r_1, r_2, r_3) \Rightarrow r_4$ *with arbitrary number of sources*
c_i2i	r_b, r_1, r_2	r_3	$r_1 \Rightarrow r_3,$ *if* $r_b =$ true $r_2 \Rightarrow r_3,$ *otherwise*
c_c2c	r_b, r_1, r_2	r_3	$r_1 \Rightarrow r_3,$ *if* $r_b =$ true $r_2 \Rightarrow r_3,$ *otherwise*

Jumps

Opcode	Sources	Results	Meaning
jump	—	r_1	$r_1 \rightarrow$ PC
jumpI	—	l_1	$l_1 \rightarrow$ PC
tbl	r_1, l_2	—	r_1 *might hold* l_2

Standard Conditional Branch Syntax

Opcode	Sources	Results	Meaning	
cmp_LT	r_1, r_2	r_3	true ⇒ r_3 false ⇒ r_3	*if* $r_1 < r_2$ *otherwise*
cmp_LE	r_1, r_2	r_3	true ⇒ r_3 false ⇒ r_3	*if* $r_1 \leq r_2$ *otherwise*
cmp_EQ	r_1, r_2	r_3	true ⇒ r_3 false ⇒ r_3	*if* $r_1 = r_2$ *otherwise*
cmp_GE	r_1, r_2	r_3	true ⇒ r_3 false ⇒ r_3	*if* $r_1 \geq r_2$ *otherwise*
cmp_GT	r_1, r_2	r_3	true ⇒ r_3 false ⇒ r_3	*if* $r_1 > r_2$ *otherwise*
cmp_NE	r_1, r_2	r_3	true ⇒ r_3 false ⇒ r_3	*if* $r_1 \neq r_2$ *otherwise*
cbr	r_1	l_2, l_3	l_2 → PC l_3 → PC	*if* $r_1 =$ true *otherwise*

Alternate Conditional Branch Syntax

Opcode	Sources	Results	Meaning	
comp	r_1, r_2	cc_3	*sets* cc_3 *to one of* {LT, LE, EQ, GE, GT, NE}	
cbr_LT	cc_1	l_2, l_3	l_2 → PC l_3 → PC	*if* $cc_3 =$ LT *otherwise*
cbr_LE	cc_1	l_2, l_3	l_2 → PC l_3 → PC	*if* $cc_3 =$ LE *otherwise*
cbr_EQ	cc_1	l_2, l_3	l_2 → PC l_3 → PC	*if* $cc_3 =$ EQ *otherwise*
cbr_GE	cc_1	l_2, l_3	l_2 → PC l_3 → PC	*if* $cc_3 =$ GE *otherwise*
cbr_GT	cc_1	l_2, l_3	l_2 → PC l_3 → PC	*if* $cc_3 =$ GT *otherwise*
cbr_NE	cc_1	l_2, l_3	l_2 → PC l_3 → PC	*if* $cc_3 =$ NE *otherwise*

Appendix B

Data Structures

B.1 INTRODUCTION

A compiler is the sum of its myriad parts. Thus, a successful compiler requires attention to many details. This appendix explores some of the algorithmic and implementation issues that arise in building a compiler. In most cases, these details would distract from the relevant discussion in the body of the text. We have gathered them together into this appendix, where they can be considered as needed.

This appendix focuses on the engineering issues that arise in the design and implementation of the infrastructure to support a compiler. The way that the compiler writer resolves these issues has a large impact on both the speed of the resulting compiler and the ease of extending and maintaining it.

In many applications, it makes sense for the programmer to rely on data-structure implementations from standard libraries. The extraordinary stability of JAVA and the increasing use of PYTHON have made this approach both practical and widespread. However, when a routine executes sufficiently often, the incremental improvement attained from custom-built data structures can be significant. On the scale of a compiler such as GCC or LLVM, the cumulative payoff may well justify the effort of implementing a custom representation for a set, a graph, a map, or another data structure.

As one example of the issues that arise in compilation, consider the size of the compiler's own data structures. The compiler cannot know the size some of those data structures until it has read the input; thus, the front end must be designed to expand the size of its data structures gracefully in order to accommodate large input files. As a corollary, however, subsequent passes in the compiler should know the approximate sizes needed for most of their internal data structures; the front end can record those sizes for the later passes. If the front end needed 10,000 names in the IR program, the compiler should not begin a later pass with a symbol table sized for 512 names. The header of any external representation of the IR should include a specification of the rough sizes of major data structures.

Similarly, the later passes of a compiler can assume that the IR program presented to them was generated by the compiler. While they should detect all errors, the error messages might be more terse than one would expect in

The arrow in ILOC is an example of this principle. It makes the flow of values explicit and eliminates one source of confusion in reading the ILOC code.

the front end. A common strategy is to build a validation pass that performs a thorough check on the IR program and can be inserted between other passes for debugging purposes, and to rely on less-strenuous error detection when not debugging the compiler. However, the compiler writers should always remember that they are the people most likely to look at the code between passes. Effort spent to make the external forms of the IR more readable often rewards the very people who invested the time and effort in it.

As a final example, consider the implementation of a sparse graph. This problem arises in optimization and in code generation; both the scheduler's dependence graph and the register allocator's interference graph tend to be sparse. The implementor might consider representing a sparse graph with a hash table. While dependence graphs for local scheduling tend to be small (limited by block size), interference graphs can grow large enough to make space efficiency a concern.

The idea is simple. Each edge is a $(source, sink)$ pair. Hashing the pairs provides a quick membership test. Edge-specific storage can be created in the hash-table entry for the edge's pair. As long as the hash function is well-behaved, this scheme produces a space-efficient representation with good asymptotic time behavior.

However, the compiler writer must take care to use a hash function that works well on the edges. If *source* and *sink* are small integers, the hash function may not have that many bits of information to use. From experience [111], hash-based interference graph representations depend heavily on the quality of the hash function.

B.2 REPRESENTING SETS

Many different problems in compilation are formulated in terms that involve sets. They arise at many points in the text, including the subset construction (Chapter 2), the construction of the canonical collection of LR(1) items (Chapter 3), data-flow analysis (Chapters 8 and 9), and worklists such as the ready queue in list scheduling (Chapter 12). In each context, the compiler writer must select an appropriate set representation. In many cases, the efficiency of the algorithm depends on careful selection of a set representation (see, for example, the *IDoms*-based algorithm in Section 9.5.2).

A fundamental difference between building a compiler and building other kinds of systems software—such as an operating system—is that many problems in compilation can be solved offline. For example, the local register-allocation algorithm in Chapter 13 is better known as Belady's MIN algorithm for offline page replacement, which has long been used as a standard against which to judge the effectiveness of online page-replacement algorithms. In the operating systems community, the algorithm is of only academic interest because it is an offline algorithm. Since the operating system cannot know what pages will be needed in the future, it cannot use an offline algorithm. In a compiler, the offline algorithm is practical because the compiler can look through the entire block before making any decisions.

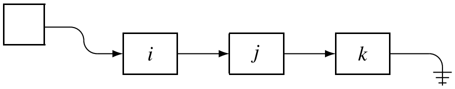
The offline nature of compilation allows the compiler writer to use a broad variety of set representations. Many representations for sets have been explored. In particular, offline computation often lets us restrict the members of a set S to a fixed-size universe U ($S \subseteq U$). This, in turn, lets us use more efficient set representations than are available in an online situation where the size of U is discovered dynamically.

Common set operations include *member*, *insert*, *delete*, *clear*, *choose-one*, *cardinality*, *forall*, *copy*, *compare*, *union*, *intersect*, *difference*, and *complement*. A specific application typically uses only a small subset of these operations. The cost of individual set operations depends on the particular representation chosen. In selecting an efficient representation for a particular application, it is important to consider how frequently each type of operation will be used. Other factors to consider include the memory requirements of the set representation and the expected sparsity of S relative to U.

The rest of this section focuses on three set representations that compiler writers often use: ordered linked lists, bit vectors, and sparse sets. The final subsection briefly discusses the role of hashing in set implementation.

B.2.1 Representing Sets as Ordered Lists

In cases in which the size of each set is small, it sometimes makes sense to use a simple linked-list representation. For a set S, this representation consists of a linked list and a pointer to the first element in the list. Each node in the list contains a representation for a single element of S and a pointer to the next element of the list. The final node on the list has its pointer set to a standard value indicating the end of the list. With a linked-list representation, the implementation can impose an order on the elements to create an ordered list. For example, an ordered linked list for the set $S = \{i, j, k\}, i < j < k$ might look, conceptually, as follows:



The elements are kept in ascending order. The size of S's representation is proportional to the number of elements in S, not the size of U. If $|S|$ is much

Operation	Ordered Linked List	Bit Vector	Sparse Set
member	$O(\|S\|)$	$O(1)$	$O(1)$
insert	$O(\|S\|)$	$O(1)$	$O(1)$
delete	$O(\|S\|)$	$O(1)$	$O(1)$
clear	$O(1)$	$O(\|U\|)$	$O(1)$
choose-one	$O(1)$	$O(\|U\|)$	$O(1)$
cardinality	$O(\|S\|)$	$O(\|U\|)$	$O(1)$
forall	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
copy	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
compare	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
union	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
intersect	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
difference	$O(\|S\|)$	$O(\|U\|)$	$O(\|S\|)$
complement	—	$O(\|U\|)$	$O(\|U\|)$

■ **FIGURE B.1** Asymptotic Time Complexities of Set Operations.

smaller than $|U|$, the savings from representing just the elements present in S may more than offset the extra cost incurred for a pointer in each element.

The list representation is particularly flexible. Because nothing in the list relies on either the size of U or the size of S, it can be used in situations in which the compiler is discovering U or S or both. Such situations arise in the front end, where the compiler acts before it has seen the entire input, and in places where it translates one name space into another, as in the construction of live ranges in a register allocator.

The table in Fig. B.1 shows the asymptotic complexities of common set operations using this representation. Most of the ordered linked list operations are $O(|S|)$ because it is necessary to walk the linked lists to perform the operations. If deallocation does not require walking the list to free the nodes for individual elements, as in a collected system or an arena-based system, *clear* takes constant time.

If the size of the universe is unknown but it can can grow reasonably large, the compiler writer can decrease the space overhead from pointers and the allocation overhead by creating nodes that hold multiple set elements. Each node has space for k elements, along with a counter that holds the number of occupied elements. Building a set of n elements then requires $\lceil \frac{n}{k} \rceil$ allocations, $\lceil \frac{n}{k} \rceil + 1$ pointers, and $\lceil \frac{n}{k} \rceil + 1$ counters, where a set implemented with single-element nodes would take n allocations and $n + 1$ pointers.

This scheme does have a cost. Insertion and deletion may need to move more data than they would with single-element nodes. Their asymptotic complexity, however, remains $O(|S|)$. A clever implementation can limit such shuffling to k elements on any insert or delete, at the cost of wasting some space.

The *IDoms* array used in the fast dominance computation (see Section 9.5.2) is a clever application of the list representation of sets to a special case. In particular, the compiler knows the size of the universe and the number of sets. The compiler also knows that, using ordered sets, the sets have the property that if $e \in S_1$ and $e \in S_2$ then every element after e in S_1 is also in S_2. Thus, the elements starting with e can be shared. By using an array representation, the element names can be used as pointers, too. Thus, a single array of n elements can represent n sparse sets. It also leads to a fast intersection operator for those sets.

B.2.2 Representing Sets as Bit Vectors

Compiler writers often use *bit vectors* to represent sets, particularly those used in data-flow analysis (see Sections 8.6.1 and 9.2). For a bounded universe U, a set $S \subseteq U$ can be represented with a bit vector of length $|U|$, called the *characteristic vector* for S. For each $i \in U$, $0 \le i < |U|$; if $i \in S$, the ith element of the characteristic vector equals one. Otherwise, the ith element is zero. For example, the characteristic vector for the set $S \subseteq U$, where $S = \{i, j, k\}$, $i < j < k$ is as follows:

0		$i-1$	i	$i+1$		$j-1$	j	$j+1$		$k-1$	k	$k+1$		$\|U\|-1$
0	…	0	1	0	…	0	1	0	…	0	1	0	…	0

The bit-vector representation always allocates enough space to represent all elements in U; thus, this representation can be used only in an application where U is known—an offline application.

The table in Fig. B.1 lists the asymptotic complexities of common set operations with this representation. Although many of the operations are $O(|U|)$, they can still be efficient if U is small. A single word holds many elements; the representation gains a constant-factor improvement over representations that need one word per element. Thus, for example, with 64-bit words, any universe of 64 or fewer elements has a single-word representation.

The compactness of the representation carries over into the speed of operations. With single-word sets, many of the set operations become single machine instructions; for example, *union* becomes a logical-or operation and *intersection* becomes a logical-and operation. Even if the sets take multiple words to represent, the number of machine instructions required to perform many of the set operations is reduced by a factor of the machine's word size.

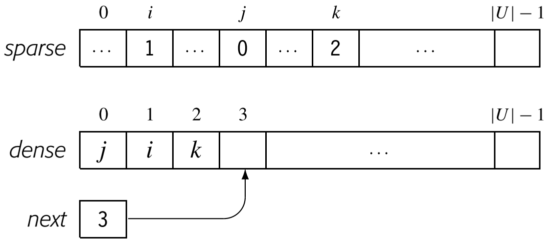
B.2.3 **Representing Sparse Sets**

For a fixed universe U and a set $S \subseteq U$, S is a sparse set if $|S|$ is much smaller than $|U|$. Some of the sets encountered in compilation are sparse. For example, the LiveOut sets used in register allocation are typically sparse. Compiler writers often use bit vectors to represent such sets, due to their efficiency in time and space. With enough sparsity, however, more time-efficient representations are possible, especially in situations in which a large percentage of the operations can be supported in either $\mathbf{O}(1)$ or $\mathbf{O}(|S|)$ time. By contrast, bit vector sets take either $\mathbf{O}(1)$ or $\mathbf{O}(|U|)$ time on these operations. If $|S|$ is smaller than $|U|$ by a factor greater than the word size, then bit vectors may be the less efficient choice.

One sparse-set representation that has these properties uses two vectors of length $|U|$ and a scalar to represent the set. The first vector, *sparse*, holds a sparse representation of the set; the other vector, *dense*, holds a dense representation of the set. The scalar, *next*, holds the index of the location in *dense* where the next new element of the set can be inserted. Of course, *next* also holds the set's cardinality.

Neither vector needs to be initialized when a sparse set is created; set membership tests ensure the validity of each entry as it is accessed. The *clear* operation simply sets *next* back to zero, its initial value. To add a new element $i \in U$ to S, the code (1) stores i in the *next* location in *dense*, (2) stores the value of *next* in the ith location in *sparse*, and (3) increments *next* so that it is the index of the next location where an element can be inserted in *dense*.

If we began with an empty sparse set S and added elements j, i, and k, in order, where $i < j < k$, the set would reach the following state:



Note that the sparse-set representation requires enough space to represent all of U. Thus, it can be used only in offline situations in which the compiler knows the size of U.

The sparse set has a concise membership test. The valid entries for an element i in *sparse* and *dense* must point to each other, so element i is in the set if and only if:

$$0 \leq \mathit{sparse}[i] < \mathit{next} \quad \text{and} \quad \mathit{dense}[\mathit{sparse}[i]] = i$$

The table in Fig. B.1 lists the asymptotic complexities of common set operations on this representation. Because this scheme includes both a sparse and a dense representation of the set, it achieves some of the advantages of each. Individual elements of the set can be accessed in $\mathbf{O}(1)$ time through *sparse*, while set operations that must traverse the set can use *dense* to obtain $\mathbf{O}(|S|)$ complexity.

Both space and time complexities should be considered when choosing between bit-vector and sparse-set representations. The sparse-set representation requires two vectors of length $|U|$ and a scalar. By contrast, a bit-vector representation requires a single bit-vector of length $|U|$. As shown in Fig. B.1, the sparse-set representation dominates the bit-vector representation in terms of asymptotic time complexity. However, because of the efficient implementations possible for bit-vector set operations, bit vectors are preferred in situations where S is not sparse. When choosing between the two representations, it is important to consider both the sparsity of the represented set and the relative frequency of the set operations employed.

B.2.4 The Role of Hash Tables

Mapping textual names, such as source-code names, compiler-generated temporary names, or virtual registers, into the internal names used in a set implementation is a critical aspect of set implementation. One effective way to create this map is with a hash table. Using the external name as the key and the internal name as the value creates an expected-case $\mathbf{O}(1)$ map.

For a name x, its *internal name* would be the position of the bit that represents a name x in a bit vector set or the index for x in a sparse set's dense vector.

B.3 IR IMPLEMENTATION

A compiler's IR is its central data structure. The IR, along with associated ancillary data structures, represents the program under translation. The compiler repeatedly traverses those structures, analyzes their contents, and rewrites those contents. Thus, the implementation of the IR plays an important role in the overall efficiency of the compiler.

Chapter 4 presented a broad overview of the kinds of intermediate representations that compilers use. This section focuses on details in the implementation of an IR. Some of the content is common sense; some of it draws on the experience of the authors and their friends in the compiler construction

When compiler writers gather, they often exchange stories about what D. R. Chase has called "stupid compiler tricks."

community. Sections B.3.1 and B.3.2 address concerns in the implementation of graphical and linear IRs, respectively.

The compiler writer should pay attention to the overall size of IR programs and to the ease with which the compiler can traverse the IR program. Both concerns affect performance. Using more memory than necessary introduces additional costs—particularly in locality and in garbage collection. In the extreme, it can limit the size of input programs that the compiler can reasonably handle. An IR that is hard to traverse introduces its own inefficiencies.

B.3.1 Graphical Intermediate Representations

Compilers use a variety of graphical IRs (see Section 4.3). Tailoring a graph's implementation to the compiler's specific needs can improve both its usability and its space efficiency. This subsection describes some of the issues that arise in tree and graph implementation.

Representing Trees

The obvious representation for a tree, in most languages, is as a collection of nodes connected by pointers. A typical implementation allocates nodes on demand, as it builds the tree. The compiler writer may choose to use multiple sizes of nodes, perhaps with different numbers of children or different data fields. Alternatively, the tree can be built from a single kind of node, allocated to fit the largest possible node.

An alternative representation might use an array of node structures. Links between nodes can be either array indices or pointers (providing the language has an *address-of* operator). This design forces a one-size-fits-all node, but is otherwise similar to the pointer-based implementation.

Each of these schemes has strengths and weaknesses. The tradeoffs must be evaluated in the context of a given project and its implementation language.

- A node and pointer scheme handles arbitrarily large trees in a natural way. An array of node structures scheme requires code to expand the array when the tree grows beyond its initial size.
- A node and pointer scheme usually requires an allocation for each node. Arena-based allocation can reduce the cost of deallocation (see the digression on page 312).
- The node and pointer scheme has locality that depends entirely on the runtime behavior of the underlying allocator. The array of node structures scheme uses a large block of consecutive memory locations, which keeps successive nodes from interfering with each other in the cache.

DECISIONS AFFECTING THE SIZE OF THE IR

The compiler writer should pay attention to the total size of the IR form of a program. Larger IRs often lead to slower compilers; in general, the compiler touches every byte of storage that it allocates. In the extreme, wasting space in the IR can cause locality problems and it can limit the size of the input programs that the compiler can easily handle.

Attention to detail in the IR design can help limit IR size.

- *Represent All Facts Compactly.* Names should be converted into small integers in the front end and translated back to strings for debugging or output. Data structures should be laid out to avoid the need for padding (see Section 5.6.5).
- *Represent Only Common Facts in the IR.* Relegate facts that are needed in just a few passes or a few operations to ancillary structures. The presence of this extra information can be encoded into another field—for example, in the sign bit of an opcode or node type.
- *Eliminate References to Dead Ancillary Structures.* If, in a managed-storage environment, the compiler allocates ancillary structures, it should overwrite any explicit pointers to those structures after they are no longer of use. Retaining such pointers keeps them alive through garbage collection.

To understand IR size, the compiler writer can build an optional reporting capability into the IR abstraction and use it, periodically, to examine IR space efficiency. In our projects, we have often discovered fields that are rarely or never used.

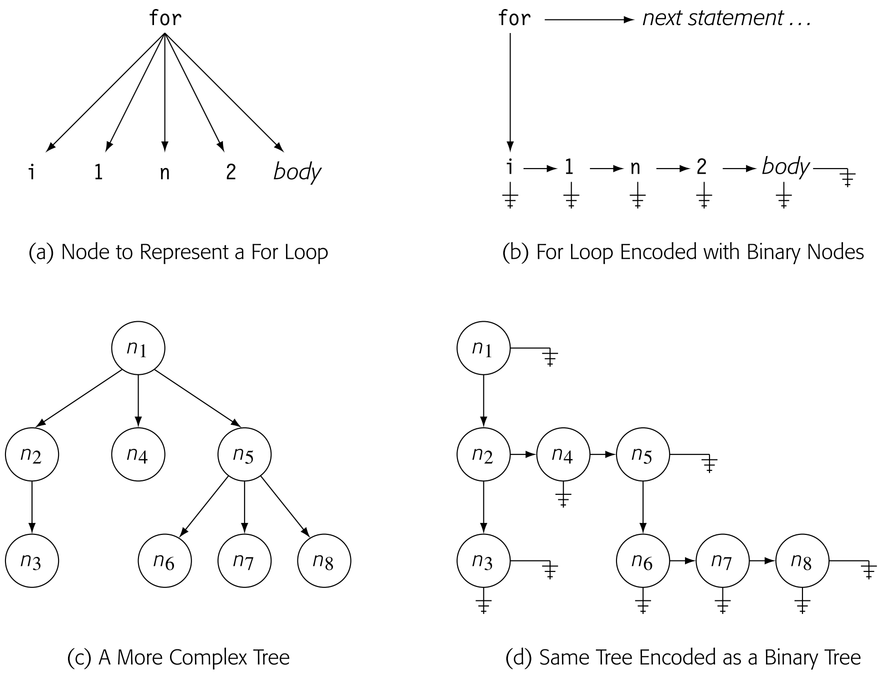
- The node and pointer scheme may be harder to debug than the array implementation. Programmers typically find array indices more intuitive than memory addresses.

The compiler writer can achieve the easy expansion of a node and pointer scheme while mitigating the allocation cost and locality issues by using a block-contiguous allocation scheme.

Block-contiguous allocation is mentioned in Chapter 4 on page 188.

A block-contiguous allocator for the tree would use a major allocator and a minor allocator. The minor allocator keeps a free list of available nodes, initialized to an empty list. When the minor allocator is called, it returns a node from the free list, unless that list is empty. The common case can be quite fast. When the free list is empty, it calls the major allocator to obtain a block of memory large enough to hold k nodes. It builds a free list of nodes from that space and then returns one of them to the caller. The tree-building code always calls the minor allocator.

Increasing k, within reason, decreases the per-node allocation cost.



(a) Node to Represent a For Loop

(b) For Loop Encoded with Binary Nodes

(c) A More Complex Tree

(d) Same Tree Encoded as a Binary Tree

■ **FIGURE B.2** Mapping Arbitrary Trees Onto Binary Trees.

Mapping Arbitrary Trees to Binary Trees

A straightforward implementation of abstract syntax trees might support nodes with many different numbers of children. For example, a typical `for` loop header

```
for i = 1 to n by 2
```

might have a node in the AST with five children, like the one shown in Fig. B.2(a). The node labeled `body` represents the subtree for the code in the body of the `for` loop.

For some constructs, no fixed number of children will work. To represent a procedure call, the AST must either custom allocate nodes based on the number of parameters or use a single child that holds a list of parameters. The same problem arises with a ϕ-function in an SSA-based IR. Using multiple node sizes complicates any code that traverses the AST; each node needs to indicate how many children it has and the traversal must contain code to visit each of those children. Using a list of parameters adds time and space overhead for allocation, construction, and traversal of the lists.

To simplify the implementation of trees, the compiler writer can map the AST's natural form—an arbitrary tree with variable arity nodes—onto a uniform binary tree in which every node has precisely two children. Under this map, a node's left-child field points to its leftmost child while its right-child field points to the next sibling at the current level in the tree. The right-children form lists of siblings.

The idea to map arbitrary trees onto binary trees is due to Knuth [240, Vol. 1].

Fig. B.2(b) shows the `for`-loop header from panel (a) encoded as a binary tree. The simplicity of the binary tree has some costs; for example, each leaf node has an explicit null pointer. Notice also that the `for` node has a pointer to the first statement after the loop; in the original version, that pointer undoubtedly occurs in the parent of the `for` node. Panels (c) and (d) show a more complex example.

Using binary trees simplifies the implementation in several ways. Uniform nodes simplify memory allocation and work well with either an arena-based allocator or a block-contiguous allocator. Code to traverse the tree is simple and uniform; it avoids the issues that arise with multiple node formats. This simplicity has its cost: additional null pointers in leaf nodes.

Arena-based allocation is discussed in the digression on page 312.

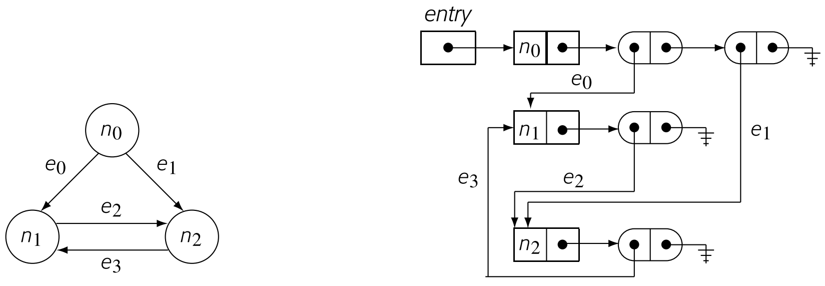
Representing Arbitrary Graphs

Compilers must also represent arbitrary graphs, such as control-flow graphs (CFGs) and dependence graphs. The same implementation issues arise in graphs as in trees. Simple implementations might use heap-allocated nodes with pointers to represent edges. Fig. B.3(a) shows a simple CFG. Clearly, it needs three nodes. The difficulty arises with the edges: how many incoming and outgoing edges does each node need? Each node could maintain a list of outgoing edges; this approach leads to an implementation similar to the one shown in Fig. B.3(b).

In Fig. B.3(b), the rectangles represent nodes, and the ovals represent edges. This representation makes it easy to walk the graph in the direction of the edges. If the compiler needs random access to the nodes, the compiler writer can add an array of node pointers, indexed by the nodes' integer names. With that addition (not shown), the graph is suitable for solving forward data-flow problems. It provides a fast means for finding all the successors of a node.

This representation does not facilitate traversing edges backward, as occurs in the solution of a backward data-flow problem. A backward traversal needs a fast predecessor operation. The compiler writer could add a list of predecessors to each node, at the cost of additional time and space.

An alternative is to represent the graph as a pair of tables: a node table and an edge table. For a given node, the node table contains the names of the



(a) Control-Flow Graph

(b) A Graphical Representation

Node Table		
Name	Succ.	Pred.
n_0	e_0	—
n_1	e_2	e_0
n_2	e_3	e_1

Edge Table				
			Next	Next
Name	Source	Sink	Succ.	Pred.
e_0	n_0	n_1	e_1	e_3
e_1	n_0	n_2	—	e_2
e_2	n_1	n_2	—	—
e_3	n_2	n_1	—	—

(c) A Tabular Representation

■ **FIGURE B.3** Tabular Representation of a CFG.

edges to its first successor and its first predecessor. The edge table has an entry for each edge; that entry contains the names of its source and sink. In addition, each edge has a field that contains the name of the next successor of its source and the next predecessor of its sink. These latter two tables instantiate, for each node, a list of its successors and a list of its predecessors.

Fig. B.3(c) shows the CFG from panel (a) in this tabular format. While the tables are not as obvious to read as the representations in panels (a) or (b), they provide quick access to successors, predecessors, and individual nodes and edges by their names (assuming that names are encoded as small integers).

The tabular representation works well for applications that traverse the graph. When the compiler makes heavy use of other operations, better representations exist. For example, a graph-coloring register allocator has two dominant graph operations: iterating over a node's neighbors and testing for the presence of an edge between two nodes (see Section 13.4.2). These operations suggest different representations; indeed, most coloring allocators use adjacency lists to iterate over neighbors quickly and a bit matrix to answer the membership question—is the edge (i, j) in the graph?

Because interference graphs are both large and sparse, space for the adjacency vectors can become an issue. Some implementations use two passes to build the graph—the first pass computes the size of each adjacency vector and the second pass builds the vectors, each with the minimal required size. Other implementations use a variant of the list representation for sets from Section B.2.1—the graph is built in a single pass, using an unordered list for the adjacency vector, with multiple edges per list node.

B.3.2 Linear Intermediate Forms

Conceptually, a linear intermediate form resembles a table or an array. The various ILOC examples throughout the book, in fact, are laid out as tables. Linear IRs have an implicit order; operations execute in order, until execution hits a branch, jump, or call. This abstract model is easy to visualize and to analyze. In practice, however, compilers use implementations that are more complex than a simple table.

A compiler manipulates a linear IR in a number of ways. It may change the values of the fields in an operation; for example, register allocation changes the register names in operations. A given pass might add, delete, or move operations. The parser adds to the end of the IR; the register allocator inserts spill code between existing operations. Dead code elimination removes operations. Lazy code motion might move operations from the middle of one block to the end of another; the scheduler reorders operations.

An implementation that uses a simple array-like structure may encounter unexpected costs on some of these manipulations. Insertions and deletions may be implemented by shuffling operations to create or reuse space. Expanding the array may entail copying the operations to a new heap object. Attention paid to such costs has a direct effect on the cost of compilation.

JAVA's `ArrayList` construct can easily exhibit $O(n^2)$ behavior if insertion and/or deletion are frequent.

The compiler writer may use different implementation strategies in different passes. For the front end, the goal might be to minimize the overhead of adding additional operations onto the end of the IR. A pass that simply performs data-flow analysis might do well with a single preallocated array of structures. The scheduler needs the ability to reorder operations; the register allocator needs to insert and delete them.

An analysis pass should know the size of the IR so that it can preallocate all the space it will require.

Implementing Operations

As with nodes in a tree, two basic options exist for a linear IR: allocating individual structures or allocating an array of structures. In either case, the structure represents a single IR operation. In the first case, the individual structures are allocated on the heap and linked using pointers (or references). In the latter case, the structures are block allocated in one or more arrays;

MITIGATION STRATEGIES FOR AN INDEX-ORDER ARRAY

The simplicity of an IR implementation that uses a single array of structures, traversed in index order, is compelling. If the compiler writer chooses this approach, she can employ strategies to mitigate some of the disadvantages.

Insertion Efficient insertions can be implemented with a `detour` operator that directs any traversal to an out-of-line code segment. To insert before operation i, the compiler creates a segment with the new operations followed by a copy of operation i, followed by a `detour` back to operation $i+1$. It then replaces operation i with a `detour` to the new code segment.

Deletion Efficient deletions are simple. The compiler can use an `ignore` operator that directs any traversal to skip to the next operation. To delete the ith operation, the compiler overwrites it with an `ignore`.

If the compiler uses these mitigation techniques, it will need to allocate additional space in the IR array to hold inserted code segments.

At the end of each pass, when the compiler writes the IR to external media for the next pass, it can linearize the `detour`s inline and remove the `ignore`s. Of course, the compiler could linearize the code in mid-pass if that was desirable.

these can be linked using either array indices (for a single array) or pointers. The advantages and disadvantages are similar to those discussed in the previous subsection.

- With linked individual structures, the compiler can easily change the traversal order. An array of structures implementation can use index-order traversals, which saves the space used by pointers; this scheme increases the cost of rearrangement.
- With an array of structures, index-order traversal should benefit from spatial locality and from the standard optimizations that improve dense linear-algebra codes, such as redundancy elimination, code motion, and strength reduction (see Chapters 8 and 10).
- With individually allocated structures, allocation costs are higher than they would be with an array. In a managed runtime, multiple smaller heap objects might affect the cost of collection.

Linear forms are amenable to multiple implementation strategies. In a pass where the IR size is known and static, the compiler writer can allocate a single array of structures large enough to hold it. If index order traversal suffices, as in a single block, the order can be maintained implicitly by the index. If the order of operations may change, the compiler writer can add pointers to each operation to create singly or doubly-linked lists and achieve arbitrary traversal orders.

If the IR size is expected to change, the compiler writer can use an arena-style allocator or a block-contiguous allocator. Such an implementation usually requires pointers to link the operations, as a transparent way to link across separately allocated arrays of structures.

Variant Node Sizes

Some operations will not fit into the standard structure for an operation. For example, the number of source operands in an SSA-form ϕ-function depends on the number of incoming edges in the control-flow graph. It might be 2; it might be 32. An implementation that uses pointers to link the individual operations can simply allocate a variant structure and link it into the IR. An implementation as an array of structures can use a `detour` operator as suggested in the digression on page 782. Of course, the code that manipulates the IR must understand all the variants and handle them.

B.4 IMPLEMENTING HASH TABLES

With the widespread availability of feature libraries in languages such as JAVA and PYTHON, most students will rely on library implementations of hash tables or maps. However, some circumstances will warrant a custom implementation. In a heavily used compiler, such as GCC or LLVM, the savings from a custom implementation may be significant when considered across all the compilations done with the compiler. In other cases, a custom implementation may take advantage of particular known properties of the set of expected keys (e.g., single character strings or pairs of small integers).

This section describes several issues that arise in the implementation of a custom hash table. Section B.4.1 describes two hash functions that, in practice, produce good results. Sections B.4.2 and B.4.3 present two widely used strategies for resolving collisions: *open-hashing* and *open-addressing*. Section B.4.4 discusses storage management issues for hash tables.

B.4.1 Choosing a Hash Function

Perhaps the most critical decision in hash-table design is the choice of a hash function. A hash function that produces a bad distribution of index values directly increases the average cost of both insertions and lookups. Fortunately, the literature documents several good hash functions, including the multiplicative hash functions described by Knuth and the universal hash functions described by Cormen et al.

WORST-CASE MAPPING BEHAVIOR

Symbol-table lookups can, and sometimes do, exhibit worst-case behavior. In the early 1980s, the authors worked on the front-end for a FORTRAN vectorizing compiler, PFC. The author of the first version of PFC's scanner used an unbalanced binary tree in its symbol table.

Unfortunately, several of the key applications used to test PFC had declaration statements that listed the variables in alphabetical order. Thus, the scanner constructed a tree in which most of the right-subtrees were empty; symbol-table lookups quickly devolved to $\mathbf{O}(n)$ time per lookup.

To remedy this situation, a graduate student was assigned to build a hash-based table. To compute the hash, the implementation broke the variable name into four-byte chunks and combined them with exclusive or. It used the resulting number, modulo the table size, as an index into the table.

Unfortunately, the student chose a table size of 2,048, so the mod operation returned the low-order 11 bits of the 32-bit word. The hash function padded short strings with blanks to reach a full word. Thus, any one or two character name produced the same hash key: the bottom 11 bits of "b̸b̸". Since the applications contained many single character variable names, symbol-table lookups again trended toward $\mathbf{O}(n)$ time per lookup. Changing the table size to a Mersenne number (2^n - 1) cured the problem. PFC used the implementation for many years with an initial table size of 2,047.

Multiplicative Hash Functions

A *multiplicative hash function* is deceptively simple. The programmer chooses a single constant *C* and uses it in the following formula:

$$h(key) = \lfloor TableSize \cdot ((C \cdot key) \bmod 1) \rfloor$$

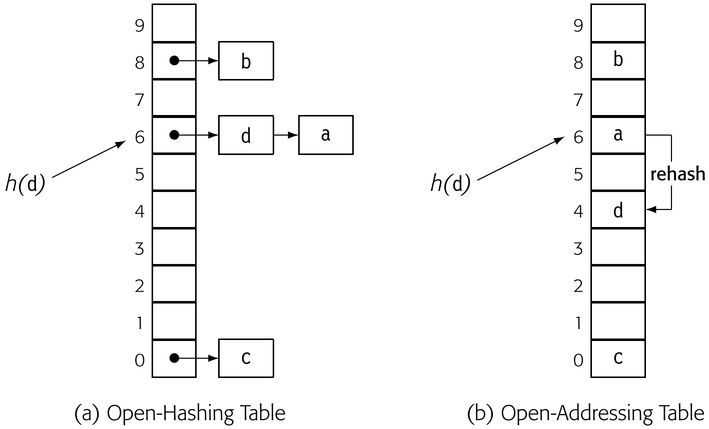
where *C* is the constant, *key* is the integer being used as a key into the table, and *TableSize* is, rather obviously, the current size of the hash table. Knuth suggests the following value for *C*:

$$0.6180339887 \approx \tfrac{\sqrt{5}-1}{2}$$

The effect of the function is to compute $C \cdot key$, take its fractional part with the mod function, and multiply the result by the size of the table.

Universal Hash Functions

To implement a *universal hash function*, the programmer designs a family of functions that are parameterized by a small set of constants. At execution



■ **FIGURE B.4** Hash-Table Organizations.

time, values for the constants are chosen at random—either using random numbers for the constants or selecting a random index into a set of previously tested constants. By varying the hash function across executions of the program, this scheme produces different distributions in different runs of the program. In a compiler, if the input program produced pathological behavior in some particular compilation, it is unlikely to produce the same behavior in subsequent compilations. To implement a universal version of the multiplicative hash function, the compiler writer can randomly generate an appropriate value for C at the start of each compilation.

The same constants are used throughout a single run of the program that uses the hash function, but the constants vary from run to run.

B.4.2 **Open Hashing**

Open hashing, also called *bucket hashing*, assumes that the hash function h produces collisions. It relies on h to partition the set of input keys into a fixed number of sets, or *buckets*. Each bucket contains a linear list of records, one record per name. *LookUp*(n) walks the linear list stored in the bucket indexed by $h(n)$ to find n. Thus, *LookUp* requires one evaluation of $h(n)$ and the traversal of a linear list. Evaluating $h(n)$ should be fast; the list traversal will take time proportional to the length of the list. For a table of size S, with N names, the cost per lookup should be roughly $\mathrm{O}(N \div S)$. As long as h distributes names fairly uniformly and the ratio of names to buckets, $N \div S$, is small, this cost approximates our goal: $\mathrm{O}(1)$ time for each access.

Fig. B.4(a) shows a small hash table implemented with this scheme. It assumes that $h(\mathtt{a}) = h(\mathtt{d}) = 6$ to create a collision. Thus, a and d occupy the same slot in the table. The list structure links them together. *Insert* should add to the front of the list for efficiency.

Open hashing has several advantages. Because it creates a new node in one of the linked lists for every inserted name, it can handle an arbitrarily large number of names without running out of space. An excessive number of entries in one bucket does not affect the cost of access in other buckets. Because the concrete representation for the set of buckets is usually an array of pointers, the overhead for increasing S is small—one pointer for each added bucket. This fact makes it less expensive to keep $N \div S$ small.

The primary drawbacks for open hashing relate directly to these advantages. Both can be managed.

1. Open hashing can be allocation intensive. Each insertion allocates a new record. When implemented on a system with heavy-weight memory allocation, the cost may be noticeable. Again, lighter-weight mechansims such as an arena-based allocator or a block-contiguous allocator can reduce the allocation costs.
2. If any set gets large, *LookUp* degrades to linear search. With a good hash function, this occurs only when N is much larger than S. The implementation should detect this problem and enlarge the array of buckets. Typically, this involves allocating a new array of buckets and reinserting each entry from the old table into the new table.

A well-implemented open hash table provides efficient access with low overhead in both space and time.

To improve the behavior of the linear search performed in a single bucket, the compiler can dynamically reorder the chain. Rivest and others [310,329] describe two effective strategies: move a node up the chain by one position on each lookup, or move it to the front of the list on each lookup. More complex schemes to organize each bucket can be used as well. However, the compiler writer should assess the total amount of time lost in traversing a bucket before investing much effort in this problem.

B.4.3 Open Addressing

Open addressing, also called *rehashing*, handles collisions by computing an alternative index for the names whose normal slot, at $h(n)$, is already occupied. In this scheme, *LookUp*(n) computes $h(n)$ and examines that slot. If the slot is empty, *LookUp* fails. If *LookUp* finds n, it succeeds. If it finds a name other than n, it uses a second function $g(n)$ to compute an increment for the search. This leads it to probe the table at $(h(n) + g(n)) \bmod S$, then at $(h(n) + 2 \times g(n)) \bmod S$, then at $(h(n) + 3 \times g(n)) \bmod S$, and so on, until it either finds n, finds an empty slot, or returns to $h(n)$ a second time. (The table is numbered from 0 to $S-1$, which ensures that mod S will return a valid table index.) If *LookUp* finds an empty slot, or it returns to $h(n)$ a second time, it fails.

Fig. B.4(b) shows a small hash table implemented with this scheme. It uses the same data as Fig. B.4(a). As before, $h(\texttt{a}) = h(\texttt{d}) = 6$, while $h(\texttt{b}) = 8$ and $h(\texttt{c}) = 0$. When d was inserted, it produced a collision with a. The secondary hash function $g(\texttt{d})$ produced 8, so *Insert* placed d at index 4 in the table $((6 + 8) \bmod 10 = 4)$. In effect, open addressing builds chains of items similar to those used in open hashing. In open addressing, however, the chains are stored directly in the table. A single table slot can be the starting point for multiple chains, each with a different increment produced by g.

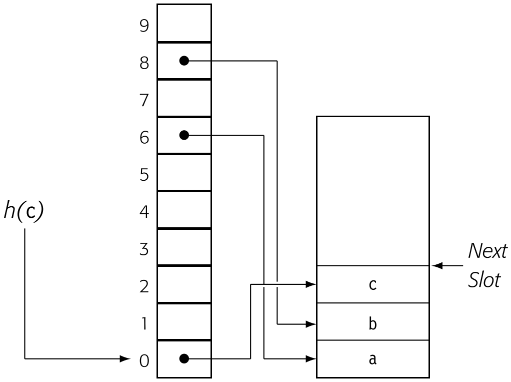
This scheme makes a subtle tradeoff of space against speed. Since each key is stored in the table, S must be larger than N. If h and g produce good distributions, then collisions are infrequent, the rehash chains stay short, and access costs stay low. Because it can recompute g inexpensively, this scheme need not store pointers to form the rehash chains—a savings of N pointers. The compiler can instead devote that space to a larger table, which may reduce the number of collisions. The primary advantage of open addressing is simple: lower access costs through shorter rehash chains.

Open addressing has two primary drawbacks. Both arise as N approaches S and the table becomes full.

1. Because rehash chains thread through the index table, a collision between n and m can interfere with a subsequent insertion of some other name p. If $h(n) = h(m)$ and $(h(m) + g(m)) \bmod S = h(p)$, then inserting n, followed by m, fills p's slot in the table. When $|S| \gg |N|$, this problem has a minor impact. As N approaches S, it can become pronounced.
2. Because S must be at least as large as N, the table must be expanded if N grows too large. (The implementation might also expand S if some chain becomes too long.) With open addressing, expansion is needed for correctness; with open hashing, expansion is a matter of efficiency.

Some implementations use a constant in place of g. While this approach eliminates the cost of computing $g(n)$, it has the downside effect that it creates a single rehash chain for each value of h. Furthermore, it merges rehash chains whenever a secondary index encounters an already occupied table slot. These two disadvantages probably outweigh the cost of computing $g(n)$. A better choice is to use the multiplicative hash function with different constants for h and g, selected at startup from a table of constants.

The table size S plays an important role in open addressing. *LookUp* must recognize when it reaches a table slot that it has already visited; otherwise, it will not halt on failure. To make this efficient, the implementation should



■ **FIGURE B.5** Stack Allocation for Records.

ensure that it eventually returns to $h(n)$. If S is a prime number, then any choice of $0 < g(n) < S$ generates a series of probes, $p_1, p_2, \ldots, p_S$, with the properties that $p_1 = p_S = h(n)$, and $p_i \neq h(n), \forall\, 1 < i < S$. That is, *LookUp* will examine every slot in the table before it returns to $h(n)$. Since the implementation may need to expand the table, it should include a table of appropriately sized prime numbers. In practice, a small set of primes will suffice, due to the realistic limits on both program size and memory available to the compiler.

B.4.4 **Storing Symbol Records**

Neither open hashing nor open addressing directly addresses the issue of how to allocate space for the information associated with each hash table entry. Assume the compiler uses a structure to hold each symbol's attributes. With open hashing, the temptation is to put these structures directly into the nodes that implement the chains. With open addressing, the temptation is to store those structures directly in the index table. Both these approaches have drawbacks. A separate stack of structures may achieve better results.

Fig. B.5 depicts this implementation. In an open-hashing implementation, the chain-pointers can be stored in the stack to avoid the need to allocate them individually. In an open-addressing implementation, the rehash chains are still implicit in the index set, preserving the space savings that motivated the idea.

Of course, the compiler writer can link the stack entries together and use either an arena-based allocator or a block-contiguous allocator to handle growth in the stack.

When the actual records are stored in a stack, they form a dense table. For heavyweight allocation, this scheme amortizes the cost of a large allocation over many records. In a collected environment, it decreases the number of objects that must be marked and collected. If the compiler writes the table to external media, it can use blocked I/O. Finally, a dense table may provide more locality when the compiler iterates over the symbols in the table—for example, when the compiler assigns storage locations.

As a final advantage, this scheme simplifies expansion of the index set. The compiler can discard the old index set, allocate a larger set, and then reinsert the records into the new table, working from the stack's bottom to its top. This process eliminates the need to have both the old and new index sets in memory at the same time. By iterating over the dense table, the compiler does not touch empty table slots.

B.5 A FLEXIBLE SYMBOL-TABLE DESIGN

Most compilers use one or more hashed symbol tables as a central repository for facts about the various names that appear in the source code, in the IR, and in the generated code. During compiler development, the set of fields in the symbol table seems to grow monotonically. Fields are added to support new passes and to transmit information between passes. When the need for a field disappears, it may or may not be removed from the symbol-table definition. As fields are added, symbol table size grows and parts of the compiler with direct access to the symbol table may need to be recompiled.

We encountered this problem in the implementation of the $\mathcal{R}^n$ and ParaScope programming environments. The experimental nature of these systems made additions and deletions of symbol-table fields a common occurence. To address the problem, we designed and built a more complex but more flexible symbol table—a *two-dimensional hash table*. It eliminated most changes to the symbol-table definition and its implementation.

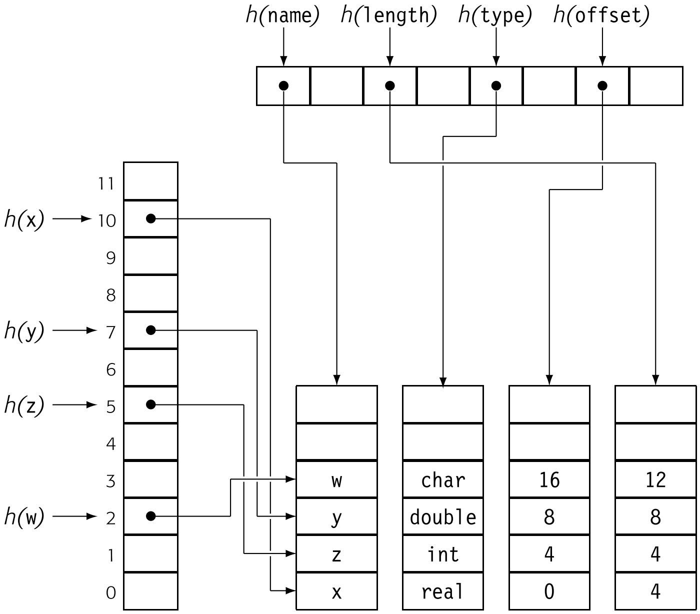
Fig. B.6 shows a conceptual drawing of such a table, populated for the C-like block shown in the margin. The implementation uses two distinct hash index tables, one for the primary key and the other for the field name, used as a secondary key. The implementation uses the secondary key to find the appropriate vector of data for the desired field. It uses the primary key to select the right slot from inside that vector. Of course, the drawing hides some complexity. Both the primary and secondary indices must be implemented as full-fledged hash tables, with provisions for collision using one of the schemes described previously.

```
{ char w[12];
  real x;
  double y;
  int z;
  ...
}
```

Declarations for Fig. B.6

While this scheme seems complex, it is not particularly expensive. Each table access requires two hash computations rather than one. The implementation need not allocate storage for a given field until a value is stored in it, which avoids the space overhead of unused fields. It allows individual developers to create and delete symbol-table fields without interfering with other programmers.

As the set of fields stabilizes, the compiler writer can replace the table with one that has fixed fields.



■ **FIGURE B.6** Two-Dimensional Hashed Symbol Table.

The implementation provided entry points for setting initial values for a field (by name), for deleting a field (by name), and for reporting statistics on field use. It allowed individual programmers to manage their own symbol-table use in a responsible and independent way, without interfering with their colleagues.

As a final issue, the implementation was abstracted with respect to specific symbol-table instances. That is, the same implementation managed tables in the parser and in other parts of the compiler.

APPENDIX NOTES

Many of the algorithms in a compiler manipulate sets, maps, tables, and graphs. The underlying implementations directly affect the space and time that those algorithms require and, ultimately, the usability of the compiler itself [63]. Textbooks cover many of the issues that this appendix brings together [5,45,119,207,240].

Our research compilers have used almost all the data structures described in this appendix. We have seen performance problems from data-structure growth in several areas.

- Abstract syntax trees, as mentioned in the sidebar in Chapter 4, can grow unreasonably large. The technique of mapping an arbitrary tree onto a binary tree simplifies the implementation and seems to keep overhead low [240].
- The tabular representation of a graph, with lists of successors and predecessors, has been reinvented many times. It works particularly well for CFGs, for which the compiler iterates over both successors and predecessors. We first used this data structure in the PFC system in 1980.
- The sets in data-flow analysis can grow quite large. Because allocation and deallocation are performance issues at that scale, we routinely use Hanson's arena-based allocator [189] for the bit-vectors.
- The size and sparsity of interference graphs makes them another area that merits careful consideration. We use an ordered-list with multiple set elements per node to keep the cost of building the graph low while managing the space overhead [111].

Symbol tables play a central role in the way that compilers store and access information. Much attention has been paid to the organization of these tables. Reorganizing lists [310,329], balanced search trees [45,119] and hashing [240, vol. 3] all play a role in making access to these tables efficient. Knuth [240, vol. 3] and Cormen [119] are standard references for high-quality hash functions.

This page intentionally left blank

Bibliography

[1] Philip S. Abrams, An APL Machine, PhD thesis, Stanford University, Stanford, CA, February 1970, Technical Report SLAC-R-114, Stanford Linear Accelerator Center, Stanford University, February 1970.

[2] Aravind Acharya, Uday Bondhugula, Albert Cohen, Polyhedral auto-transformation with no integer linear programming, in: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 53 (4) (June 2018) 529–542.

[3] Alfred V. Aho, Mahadevan Ganapathi, Steven W.K. Tjiang, Code generation using tree matching and dynamic programming, ACM Transactions on Programming Languages and Systems (TOPLAS) 11 (4) (October 1989) 491–516.

[4] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, On finding lowest common ancestors in trees, in: Conference Record of the Fifth Annual ACM Symposium on Theory of Computing (STOC ’73), Austin, TX, May 1973, pp. 253–265.

[5] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[6] Alfred V. Aho, Stephen C. Johnson, Optimal code generation for expression trees, Journal of the ACM 23 (3) (July 1976) 488–501.

[7] Alfred V. Aho, Stephen C. Johnson, Jeffrey D. Ullman, Code generation for expressions with common subexpressions, in: Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, GA, January 1976, pp. 19–31.

[8] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading, MA, 1986.

[9] Alfred V. Aho, Jeffrey D. Ullman, The Theory of Parsing, Translation, and Compiling, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[10] Philippe Aigrain, Susan L. Graham, Robert R. Henry, Marshall Kirk McKusick, Eduardo Pelegrí-Llopart, Experience with a Graham-Glanville style code generator, in: Proceedings of the ACM SIGPLAN ’84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 13–24.

[11] Alexander Aiken, Alexandru Nicolau, Optimal loop parallelization, in: Proceedings of the ACM SIGPLAN ’88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23 (7) (July 1988) 308–317.

[12] Frances E. Allen, Program optimization, in: Mark I. Halprin, Christopher J. Shaw (Eds.), Annual Review in Automatic Programming, vol. 5, Pergamon Press, Oxford, England, 1969, pp. 239–307.

[13] Frances E. Allen, Control flow analysis, in: Proceedings of a Symposium on Compiler Optimization, ACM SIGPLAN Notices 5 (7) (July 1970) 1–19.

[14] Frances E. Allen, A basis for program optimization, in: Foundations and Systems, in: Information Processing: Proceedings of IFIP Congress 1971, vol. 1, Ljubljana, Yugoslavia, August 1971, North-Holland Publishing Company, Amsterdam, 1972, pp. 385–390.

[15] Frances E. Allen, The history of language processor technology in IBM, IBM Journal of Research and Development 25 (5) (September 1981) 535–548.

[16] Frances E. Allen, Private communication. Dr. Allen noted that Beatty described live analysis in a September 1968 document titled “Optimization Methods for Highly Parallel, Multiregister Machines,” April 2009.

[17] Frances E. Allen, John Cocke, A catalogue of optimizing transformations, in: R. Rustin (Ed.), Design and Optimization of Compilers, Prentice-Hall, Englewood Cliffs, NJ, June 1972, pp. 1–30.

[18] Frances E. Allen, John Cocke, Graph-theoretic constructs for program flow analysis, Technical Report RC 3923 (17789), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, July 1972.

[19] Frances E. Allen, John Cocke, A program data flow analysis procedure, Communications of the ACM 19 (3) (March 1976) 137–147.

[20] Frances E. Allen, John Cocke, Ken Kennedy, Reduction of operator strength, in: Steven S. Muchnick, Neil D. Jones (Eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1981, chapter 3, pp. 79–101.

[21] John R. Allen, Ken Kennedy, Optimizing Compilers for Modern Architectures, Morgan Kaufmann, San Francisco, CA, October 2001.

[22] Bowen Alpern, Fred B. Schneider, Verifying temporal properties without temporal logic, ACM Transactions on Programming Languages and Systems (TOPLAS) 11 (1) (January 1989) 147–167.

[23] Bowen Alpern, Mark N. Wegman, F. Kenneth Zadeck, Detecting equality of variables in programs, in: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, CA, January 1988, pp. 1–11.

[24] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, Mikkel Thorup, Dominators in linear time, SIAM Journal on Computing 28 (6) (June 1999) 2117–2132.

[25] Marc A. Auslander, Martin E. Hopkins, An overview of the PL.8 compiler, in: Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 22–31.

[26] Andrew Ayers, Robert Gottlieb, Richard Schooler, Aggressive inlining, in: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 134–145.

[27] John W. Backus, The history of FORTRAN I, II, and III, in: Richard L. Wexelblat (Ed.), History of Programming Languages, Academic Press, New York, NY, 1981, pp. 25–45.

[28] John W. Backus, R.J. Beeber, S. Best, R. Goldberg, L.M. Haibt, H.L. Herrick, R.A. Nelson, D. Sayre, P.B. Sheridan, H. Stern, I. Ziller, R.A. Hughes, R. Nutt, The FORTRAN automatic coding system, in: Proceedings of the Western Joint Computer Conference, Institute of Radio Engineers, New York, NY, February 1957, pp. 188–198.

[29] David F. Bacon, Susan L. Graham, Oliver J. Sharp, Compiler transformations for high-performance computing, ACM Computing Surveys 26 (4) (1994) 345–420.

[30] Jean-Loup Baer, D.P. Bovet, Compilation of arithmetic expressions for parallel computations, in: Mathematics, Software, Volume 1 of Information Processing 68: Proceedings of IFIP Congress 1968, Edinburgh, UK, August 1968, North-Holland Publishing Company, Amsterdam, 1968, pp. 340–346.

[31] John T. Bagwell Jr., Local optimizations, in: Proceedings of a Symposium on Compiler Optimization, ACM SIGPLAN Notices 5 (7) (July 1970) 52–66.

[32] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, Dynamo: A transparent dynamic optimization system, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 35 (5) (May 2000) 1–12.

[33] Vasanth Bala, Norman Rubin, Efficient instruction scheduling using finite state automata, International Journal of Parallel Programming 25 (2) (April 1997) 53–82.

[34] J. Eugene Ball, Predicting the effects of optimization on a procedure body, in: Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction, ACM SIGPLAN Notices 14 (8) (August 1979) 214–220.

[35] John Banning, An efficient way to find side effects of procedure calls and aliases of variables, in: Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, TX, January 1979, pp. 29–41.

[36] William A. Barrett, John D. Couch, Compiler Construction: Theory and Practice, Science Research Associates, Inc., Chicago, IL, 1979.

[37] Jeffrey M. Barth, An interprocedural data flow analysis algorithm, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, CA, January 1977, pp. 119–131.

[38] Alan M. Bauer, Harry J. Saal, Does APL really need run-time checking?, Software—Practice and Experience 4 (2) (1974) 129–138.

[39] Laszlo A. Belady, A study of replacement algorithms for a virtual storage computer, IBM Systems Journal 5 (2) (1966) 78–101.

[40] Peter Bergner, Peter Dahl, David Engebretsen, Matthew T. O'Keefe, Spill code minimization via interference region spilling, in: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 287–295.

[41] David Bernstein, Dina Q. Goldin, Martin Charles Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, Ron Y. Pinter, Spill code minimization techniques for optimizing compilers, in: Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 24 (7) (July 1989) 258–263.

[42] David Bernstein, Michael Rodeh, Global instruction scheduling for superscalar machines, in: Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 26 (6) (June 1991) 241–255.

[43] Robert L. Bernstein, Producing good code for the case statement, Software—Practice and Experience 15 (10) (October 1985) 1021–1024.

[44] David A. Berson, Rajiv Gupta, Mary Lou Soffa, Integrated instruction scheduling and register allocation techniques, in: Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing, in: Lecture Notes in Computer Science (LNCS), vol. 1656, Springer-Verlag, Berlin-Heidelberg, Germany, 1998, pp. 247–262.

[45] Andrew Binstock, John Rex, Practical Algorithms for Programmers, Addison-Wesley, Reading, MA, 1995.

[46] Peter L. Bird, An implementation of a code generator specification language for table driven code generators, in: Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 44–55.

[47] Rastislav Bodík, Rajiv Gupta, Mary Lou Soffa, Complete removal of redundant expressions, in: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 33 (5) (May 1998) 1–14.

[48] Hans-Juergen Boehm, Space efficient conservative garbage collection, in: Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 28 (6) (June 1993) 197–206.

[49] Hans-Juergen Boehm, Alan Demers, Implementing Russell, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 186–195.

[50] Hans-Juergen Boehm, Mark Weiser, Garbage collection in an uncooperative environment, Software—Practice and Experience 18 (9) (September 1988) 807–820.

[51] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, Christophe Guillon, Revisiting out-of-SSA translation for correctness, code quality and efficiency, in: Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09), Seattle, WA, March 2009, pp. 114–125.

[52] Uday Bondhugula, Albert Hartono, J. Ramanujam, P. Sadayappan, A practical automatic polyhedral parallelizer and locality optimizer, in: Proceedings of the 29th

ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 43 (6) (June 2008) 101–113.

[53] Florent Bouchez, A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases, PhD thesis, École Normale Supérieur de Lyon, Lyon, France, April 2009.

[54] David G. Bradlee, Susan J. Eggers, Robert R. Henry, Integrating register allocation and instruction scheduling for RISCs, in: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), ACM SIGPLAN Notices 26 (4) (April 1991) 122–131.

[55] Preston Briggs, Register Allocation via Graph Coloring, PhD thesis, Department of Computer Science, Rice University, Houston, TX, April 1992, Technical Report TR92-183, Computer Science Department, Rice University, 1992.

[56] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, L. Taylor Simpson, Practical improvements to the construction and destruction of static single assignment form, Software—Practice and Experience 28 (8) (July 1998) 859–881.

[57] Preston Briggs, Keith D. Cooper, Ken Kennedy, Linda Torczon, Coloring heuristics for register allocation, in: Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 24 (7) (July 1989) 275–284.

[58] Preston Briggs, Keith D. Cooper, Ken Kennedy, Linda Torczon, Digital computer register allocation and code spilling using interference graph coloring, United States Patent 5,249,295, March 1993.

[59] Preston Briggs, Keith D. Cooper, L. Taylor Simpson, Value numbering, Software—Practice and Experience 27 (6) (June 1997) 701–724.

[60] Preston Briggs, Keith D. Cooper, Linda Torczon, Coloring register pairs, ACM Letters on Programming Languages and Systems (LOPLAS) 1 (1) (March 1992) 3–13.

[61] Preston Briggs, Keith D. Cooper, Linda Torczon, Rematerialization, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 27 (7) (July 1992) 311–321.

[62] Preston Briggs, Keith D. Cooper, Linda Torczon, Improvements to graph coloring register allocation, ACM Transactions on Programming Languages and Systems (TOPLAS) 16 (3) (May 1994) 428–455.

[63] Preston Briggs, Linda Torczon, An efficient representation for sparse sets, ACM Letters on Programming Languages and Systems (LOPLAS) 2 (1–4) (March–December 1993) 59–69.

[64] Philip Brisk, Foad Dabiri, Jamie Macbeth, Majid Sarrafzadeh, Polynomial time graph coloring register allocation, in: Fourteenth International Workshop on Logic and Synthesis, Lake Arrowhead, CA, June 2005, pp. 447–454.

[65] Klaus Brouwer, Wolfgang Gellerich, Erhard Ploedereder, Myths and facts about the efficient implementation of finite automata and lexical analysis, in: Proceedings of the Seventh International Conference on Compiler Construction (CC '98), in: Lecture Notes in Computer Science (LNCS), vol. 1383, Springer-Verlag, Heidelberg, Germany, 1998, pp. 1–15.

[66] Janusz A. Brzozowski, Canonical regular expressions and minimal state graphs for definite events, in: Mathematical Theory of Automata, in: MRI Symposia Series, vol. 12, Polytechnic Press, Polytechnic Institute of Brooklyn, New York, NY, 1962, pp. 529–561.

[67] Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, Jeffery R. Westbrook, Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC), Dallas, TX, 1998, pp. 279–288.

[68] Michael Burke, An interval-based approach to exhaustive and incremental interprocedural data-flow analysis, ACM Transactions on Programming Languages and Systems (TOPLAS) 12 (3) (July 1990) 341–395.

[69] Michael Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Serrano, V.C. Sreedhar, Harini Srinivasan, The Jalapeño dynamic optimizing compiler for Java, in: Proceedings of the ACM 1999 Conference on Java Grande, San Francisco, CA, June 1999, pp. 129–141.

[70] Michael Burke, Linda Torczon, Interprocedural optimization: Eliminating unnecessary recompilation, ACM Transactions on Programming Languages and Systems (TOPLAS) 15 (3) (July 1993) 367–399.

[71] Jiazhen Cai, Robert Paige, Using multiset discrimination to solve language processing problems without hashing, Theoretical Computer Science 145 (1–2) (1995) 189–228.

[72] Brad Calder, Dirk Grunwald, Reducing branch costs via branch alignment, in: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), ACM SIGPLAN Notices 29 (11) (November 1994) 242–251.

[73] David Callahan, Steve Carr, Ken Kennedy, Improving register allocation for subscripted variables, in: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 25 (6) (June 1990) 53–65.

[74] David Callahan, Keith D. Cooper, Ken Kennedy, Linda Torczon, Interprocedural constant propagation, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 152–161.

[75] David Callahan, Brian Koblenz, Register allocation via hierarchical graph coloring, in: Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 26 (6) (June 1991) 192–203.

[76] Luca Cardelli, Type systems, in: Allen B. Tucker Jr. (Ed.), The Computer Science and Engineering Handbook, CRC Press, Boca Raton, FL, December 1996, chapter 103, pp. 2208–2236.

[77] Steve Carr, Ken Kennedy, Scalar replacement in the presence of conditional control flow, Software—Practice and Experience 24 (1) (January 1994) 51–77.

[78] Roderic G.G. Cattell, Automatic derivation of code generators from machine descriptions, ACM Transactions on Programming Languages and Systems (TOPLAS) 2 (2) (April 1980) 173–190.

[79] Roderic G.G. Cattell, Joseph M. Newcomer, Bruce W. Leverett, Code generation in a machine-independent compiler, in: Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction, ACM SIGPLAN Notices 14 (8) (August 1979) 65–75.

[80] Gregory J. Chaitin, Register allocation and spilling via graph coloring, in: Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 98–105.

[81] Gregory J. Chaitin, Register allocation and spilling via graph coloring, United States Patent 4,571,678, February 1986.

[82] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, Peter W. Markstein, Register allocation via coloring, Computer Languages 6 (1) (January 1981) 47–57.

[83] David R. Chase, An improvement to bottom-up tree pattern matching, in: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 1987, pp. 168–177.

[84] David R. Chase, Mark Wegman, F. Kenneth Zadeck, Analysis of pointers and structures, in: Proceedings of the ACM SIGPLAN '90 Conference on Programming

Language Design and Implementation, ACM SIGPLAN Notices 25 (6) (June 1990) 296–310.

[85] J. Bradley Chen, Bradley D.D. Leupen, Improving instruction locality with just-in-time code layout, in: Proceedings of the First USENIX Windows NT Workshop, Seattle, WA, August 1997, pp. 25–32.

[86] C.J. Cheney, A nonrecursive list compacting algorithm, Communications of the ACM 13 (11) (November 1970) 677–678.

[87] Jong-Deok Choi, Michael Burke, Paul R. Carini, Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects, in: Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, SC, January 1993, pp. 232–245.

[88] Frederick C. Chow, A Portable Machine-Independent Global Optimizer—Design and Measurements, PhD thesis, Department of Electrical Engineering, Stanford University, December 1983, Technical Report CSL-TR-83-254, Computer Systems Laboratory, Stanford University, Palo Alto, CA, USA, December 1983.

[89] Cliff Click, Combining Analyses, Combining Optimizations, PhD thesis, Department of Computer Science, Rice University, Houston, TX, February 1995, Technical Report TR95-252, Computer Science Department, Rice University, 1995.

[90] Cliff Click, Global code motion/global value numbering, in: Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 30 (6) (June 1995) 246–257.

[91] Cliff Click, Keith D. Cooper, Combining analyses, combining optimizations, ACM Transactions on Programming Languages and Systems (TOPLAS) 17 (2) (March 1995) 181–196.

[92] Cliff Click, Michael Paleczny, A simple graph-based intermediate representation, in: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations, ACM SIGPLAN Notices 30 (3) (March 1995) 35–49.

[93] Cliff Click, Michael Paleczny, Christopher Vick, Interference graph trimming (unpublished), Technical report available on ResearchGate, May 2004.

[94] John Cocke, Global common subexpression elimination, in: Proceedings of a Symposium on Compiler Optimization, ACM SIGPLAN Notices 5 (7) (July 1970) 20–24.

[95] John Cocke, Ken Kennedy, An algorithm for reduction of operator strength, Communications of the ACM 20 (11) (November 1977) 850–856.

[96] John Cocke, Peter W. Markstein, Measurement of program improvement algorithms, in: Simon H. Lavington (Ed.), Information Processing 80, Proceedings of IFIP Congress 80, North Holland, Amsterdam, Netherlands, 1980, pp. 221–228.

[97] John Cocke, Peter W. Markstein, Strength reduction for division and modulo with application to accessing a multilevel store, IBM Journal of Research and Development 24 (6) (1980) 692–694.

[98] John Cocke, Jacob T. Schwartz, Programming languages and their compilers: Preliminary notes, Technical Report, Courant Institute of Mathematical Sciences, New York University, New York, NY, 1970.

[99] Jacques Cohen, Garbage collection of linked structures, ACM Computing Surveys 13 (3) (September 1981) 341–367.

[100] Robert Cohn, P. Geoffrey Lowney, Hot cold optimization of large Windows/NT applications, in: Proceedings of the Twenty-Ninth IEEE/ACM Annual International Symposium on Microarchitecture (MICRO-29), Paris, France, December 1996, pp. 80–89.

[101] Stephanie Coleman, Kathryn S. McKinley, Tile size selection using cache organization and data layout, in: Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 30 (6) (June 1995) 279–290.

[102] George E. Collins, A method for overlapping and erasure of lists, Communications of the ACM 3 (12) (December 1960) 655–657.

[103] Melvin E. Conway, Design of a separable transition diagram compiler, Communications of the ACM 6 (7) (July 1963) 396–408.

[104] Richard W. Conway, Thomas R. Wilcox, Design and implementation of a diagnostic compiler for PL/I, Communications of the ACM 16 (3) (March 1973) 169–179.

[105] Katherine Coons, Warren Hunt, Bertrand A. Maher, Doug Burger, Kathryn S. McKinley, Optimal Huffman tree-height reduction for instruction-level parallelism, Technical Report TR-08-34, Department of Computer Science, The University of Texas at Austin, Austin, TX, August 2008.

[106] Keith D. Cooper, Anshuman Dasgupta, Jason Eckhardt, Revisiting graph coloring register allocation: A study of the Chaitin-Briggs and Callahan-Koblenz algorithms, in: Proceedings of the Eighteenth International Conference on Languages and Compilers for Parallel Computing (LCPC ’05), in: Lecture Notes in Computer Science (LNCS), vol. 4339, Springer-Verlag, Berlin, Heidelberg, Germany, 2006, pp. 1–16.

[107] Keith D. Cooper, Jason Eckhardt, Improved passive splitting, in: Proceedings of the 2005 International Conference on Programming Languages and Compilers, CSREA Press, Las Vegas, NV, June 2005, pp. 115–122.

[108] Keith D. Cooper, Mary W. Hall, Linda Torczon, An experiment with inline substitution, Software—Practice and Experience 21 (6) (June 1991) 581–601.

[109] Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, An empirical study of iterative data-flow analysis, in: Proceedings of the Fifteenth International Conference on Computing (CIC’06), IEEE Computer Society, Washington, D.C., 2006, pp. 266–276.

[110] Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, A simple, fast dominance algorithm, Technical Report TR-06-38870, Department of Computer Science, Rice University, Houston, TX, January 2006.

[111] Keith D. Cooper, Timothy J. Harvey, Linda Torczon, How to build an interference graph, Software—Practice and Experience 28 (4) (April 1998) 425–444.

[112] Keith D. Cooper, Ken Kennedy, Interprocedural side-effect analysis in linear time, in: Proceedings of the ACM SIGPLAN ’88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23 (7) (July 1988) 57–66.

[113] Keith D. Cooper, Ken Kennedy, Fast interprocedural alias analysis, in: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, TX, January 1989, pp. 49–59.

[114] Keith D. Cooper, Ken Kennedy, Linda Torczon, The impact of interprocedural analysis and optimization in the $\mathcal{R}^n$ programming environment, ACM Transactions on Programming Languages and Systems (TOPLAS) 8 (4) (October 1986) 491–523.

[115] Keith D. Cooper, Philip J. Schielke, Non-local instruction scheduling with limited code growth, in: F. Mueller, A. Bestavros (Eds.), Proceedings of the 1998 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), in: Lecture Notes in Computer Science (LNCS), vol. 1474, Springer-Verlag, Heidelberg, Germany, 1998, pp. 193–207.

[116] Keith D. Cooper, L. Taylor Simpson, Live range splitting in a graph coloring register allocator, in: Proceedings of the Seventh International Compiler Construction Conference (CC ’98), in: Lecture Notes in Computer Science (LNCS), vol. 1383, Springer-Verlag, Heidelberg, Germany, 1998, pp. 174–187.

[117] Keith D. Cooper, L. Taylor Simpson, Christopher A. Vick, Operator strength reduction, ACM Transactions on Programming Languages and Systems (TOPLAS) 23 (5) (September 2001) 603–625.

[118] Keith D. Cooper, Todd Waterman, Understanding energy consumption on the C62x, in: Proceedings of the 2002 Workshop on Compilers and Operating Systems for Low Power, Charlottesville, VA, September 2002, pp. 4-1–4-8.

[119] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1992.

[120] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (4) (October 1991) 451–490.

[121] Ron Cytron, Andy Lowry, F. Kenneth Zadeck, Code motion of control structures in high-level languages, in: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, January 1986, pp. 70–85.

[122] Jan Daciuk, Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings, in: Proceedings of the Seventh International Conference on Implementation and Application of Automata (CIAA 2002), in: Lecture Notes in Computer Science (LNCS), vol. 2608, Springer, Berlin, Heidelberg, 2003, pp. 255–261.

[123] Manuvir Das, Unification-based pointer analysis with directional assignments, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 35 (5) (May 2000) 35–46.

[124] Anshuman Dasgupta, Tailoring Traditional Optimizations for Runtime Compilation, PhD thesis, Department of Computer Science, Rice University, Houston, TX, September 2006.

[125] Jack W. Davidson, Christopher W. Fraser, The design and application of a retargetable peephole optimizer, ACM Transactions on Programming Languages and Systems (TOPLAS) 2 (2) (April 1980) 191–202.

[126] Jack W. Davidson, Christopher W. Fraser, Automatic generation of peephole optimizations, in: Proceedings of the ACM SIGPLAN ’84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 111–116.

[127] Jack W. Davidson, Christopher W. Fraser, Register allocation and exhaustive peephole optimization, Software—Practice and Experience 14 (9) (September 1984) 857–865.

[128] Jack W. Davidson, Christopher W. Fraser, Automatic inference and fast interpretation of peephole optimization rules, Software—Practice and Experience 17 (11) (November 1987) 801–812.

[129] Jack W. Davidson, Ann M. Holler, A study of a C function inliner, Software—Practice and Experience 18 (8) (August 1988) 775–790.

[130] Jack W. Davidson, Sanjay Jinturkar, Aggressive loop unrolling in a retargetable, optimizing compiler, in: Proceedings of the Sixth International Conference on Compiler Construction (CC ’96), in: Lecture Notes in Computer Science (LNCS), vol. 1060, Springer, Berlin, Heidelberg, April 1996, pp. 59–73.

[131] Alan J. Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, Scott Shenker, Combining generational and conservative garbage collection: Framework and implementations, in: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, CA, January 1990, pp. 261–269.

[132] Frank DeRemer, Simple LR(k) grammars, Communications of the ACM 14 (7) (July 1971) 453–460.

[133] Frank DeRemer, Thomas J. Pennello, Efficient computation of LALR(1) lookahead sets, in: Proceedings of the ACM SIGPLAN ’79 Symposium on Compiler Construction, ACM SIGPLAN Notices 14 (8) (August 1979) 176–187.

[134] Alain Deutsch, Interprocedural may-alias analysis for pointers: Beyond k-limiting, in: Proceedings of the ACM SIGPLAN ’94 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 29 (6) (June 1994) 230–241.

[135] L. Peter Deutsch, An Interactive Program Verifier, PhD thesis, Computer Science Department, University of California, Berkeley, Berkeley, CA, 1973, Technical Report CSL-73-1, Xerox Palo Alto Research, May 1973.

[136] L. Peter Deutsch, Daniel G. Bobrow, An efficient, incremental, automatic, garbage collector, Communications of the ACM 19 (9) (September 1976) 522–526.

[137] L. Peter Deutsch, Allan M. Schiffman, Efficient implementation of the Smalltalk-80 system, in: Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, UT, January 1984, pp. 297–302.

[138] Dhananjay M. Dhamdhere, On algorithms for operator strength reduction, Communications of the ACM 22 (5) (May 1979) 311–312.

[139] Dhananjay M. Dhamdhere, A fast algorithm for code movement optimisation, ACM SIGPLAN Notices 23 (10) (October 1988) 172–180.

[140] Dhananjay M. Dhamdhere, A new algorithm for composite hoisting and strength reduction, International Journal of Computer Mathematics 27 (1) (1989) 1–14.

[141] Dhananjay M. Dhamdhere, Practical adaptation of the global optimization algorithm of Morel and Renvoise, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (2) (April 1991) 291–294.

[142] Dhananjay M. Dhamdhere, J.R. Isaac, A composite algorithm for strength reduction and code movement optimization, International Journal of Computer and Information Sciences 9 (3) (June 1980) 243–273.

[143] Michael K. Donegan, Robert E. Noonan, Stefan Feyock, A code generator generator language, in: Proceedings of the ACM SIGPLAN '79 Symposium on Compiler Construction, ACM SIGPLAN Notices 14 (8) (August 1979) 58–64.

[144] Karl-Heinz Drechsler, Manfred P. Stadel, A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies," ACM Transactions on Programming Languages and Systems (TOPLAS) 10 (4) (October 1988) 635–640.

[145] Karl-Heinz Drechsler, Manfred P. Stadel, A variation of Knoop, Rüthing, and Steffen's "Lazy code motion," ACM SIGPLAN Notices 28 (5) (May 1993) 29–38.

[146] Jay Earley, An efficient context-free parsing algorithm, Communications of the ACM 13 (2) (February 1970) 94–102.

[147] Kemal Ebcioğlu, Toshio Nakatani, A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture, in: Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing (LCPC '89), Pitman Publishing, London, UK, 1990, pp. 213–229.

[148] John R. Ellis, Bulldog: A Compiler for VLIW Architectures, The MIT Press, Cambridge, MA, 1986.

[149] Maryam Emami, Rakesh Ghiya, Laurie J. Hendren, Context-sensitive interprocedural points-to analysis in the presence of function pointers, in: Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 29 (6) (June 1994) 242–256.

[150] Andrei P. Ershov, On programming of arithmetic expressions, Communications of the ACM 1 (8) (August 1958) 3–6 (The figures appear in volume 1, number 9, page 16).

[151] Andrei P. Ershov, Reduction of the problem of memory allocation in programming to the problem of coloring the vertices of graphs, Soviet Mathematics 3 (1962) 163–165, Originally published in Doklady Akademii Nauk SSSR 142 (4) (1962).

[152] Andrei P. Ershov, Alpha—An automatic programming system of high efficiency, Journal of the ACM 13 (1) (January 1966) 17–24.

[153] M. Anton Ertl, Optimal code selection in DAGs, in: Proceedings of the Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, January 1999, pp. 242–249.

[154] Robert R. Fenichel, Jerome C. Yochelson, A LISP garbage-collector for virtual-memory computer systems, Communications of the ACM 12 (11) (November 1969) 611–612.

[155] Jeanne Ferrante, Karl J. Ottenstein, Joe D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems (TOPLAS) 9 (3) (July 1987) 319–349.

[156] Charles N. Fischer, Richard J. LeBlanc Jr., The implementation of run-time diagnostics in Pascal, IEEE Transactions on Software Engineering SE-6 (4) (1980) 313–319.

[157] Charles N. Fischer, Richard J. LeBlanc Jr., Crafting a Compiler with C, Benjamin/Cummings, Redwood City, CA, 1991.

[158] Joseph A. Fisher, Trace scheduling: A technique for global microcode compaction, IEEE Transactions on Computers C-30 (7) (July 1981) 478–490.

[159] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, Alexandru Nicolau, Parallel processing: A smart compiler and a dumb machine, in: Proceedings of the ACM SIGPLAN ’84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 37–47.

[160] Robert W. Floyd, An algorithm for coding efficient arithmetic expressions, Communications of the ACM 4 (1) (January 1961) 42–51.

[161] J.M. Foster, A syntax improving program, Computer Journal 11 (1) (May 1968) 31–34.

[162] Christopher W. Fraser, David R. Hanson, Todd A. Proebsting, Engineering a simple, efficient code generator generator, ACM Letters on Programming Languages and Systems (LOPLAS) 1 (3) (September 1992) 213–226.

[163] Christopher W. Fraser, Robert R. Henry, Hard-coding bottom-up code generation tables to save time and space, Software—Practice and Experience 21 (1) (January 1991) 1–12.

[164] Christopher W. Fraser, Alan L. Wendt, Integrating code generation and optimization, in: Proceedings of the ACM SIGPLAN ’86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 242–248.

[165] Christopher W. Fraser, Alan L. Wendt, Automatic generation of fast optimizing code generators, in: Proceedings of the ACM SIGPLAN ’88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23 (7) (July 1988) 79–84.

[166] Mahadevan Ganapathi, Charles N. Fischer, Description-driven code generation using attribute grammars, in: Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM, January 1982, pp. 108–119.

[167] Karthik Gargi, A sparse algorithm for predicated global value numbering, in: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 37 (5) (May 2002) 45–56.

[168] George Lal, Andrew W. Appel, Iterated register coalescing, in: Proceedings of the Twenty-Third ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, January 1996, pp. 208–218.

[169] Phillip B. Gibbons, Steven S. Muchnick, Efficient instruction scheduling for a pipelined architecture, in: Proceedings of the ACM SIGPLAN ’86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 11–16.

[170] R. Steven Glanville, Susan L. Graham, A new method for compiler code generation, in: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, AZ, January 1978, pp. 231–240.

[171] Nikolas Gloy, Michael D. Smith, Procedure placement using temporal-ordering information, ACM Transactions on Programming Languages and Systems (TOPLAS) 21 (5) (September 1999) 977–1027.

[172] Adele Goldberg, David Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.

[173] James R. Goodman, Wei-Chung Hsu, Code scheduling and register allocation in large basic blocks, in: Proceedings of the Second International Conference on Supercomputing, Saint Malo, France, July 1988, pp. 442–452.

[174] Eiichi Goto, Monocopy and associative operations in extended Lisp, Technical Report 74-03, University of Tokyo, Tokyo, Japan, May 1974.

[175] Susan L. Graham, Table-driven code generation, IEEE Computer 13 (8) (August 1980) 25–34.

[176] Susan L. Graham, Michael A. Harrison, Walter L. Ruzzo, An improved context-free recognizer, ACM Transactions on Programming Languages and Systems (TOPLAS) 2 (3) (July 1980) 415–462.

[177] Susan L. Graham, Robert R. Henry, Robert A. Schulman, An experiment in table driven code generation, in: Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 32–43.

[178] Susan L. Graham, Mark Wegman, A fast and usually linear algorithm for global flow analysis, in: Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, CA, January 1975, pp. 22–34.

[179] Susan L. Graham, Mark Wegman, A fast and usually linear algorithm for global flow analysis, Journal of the ACM 23 (1) (1976) 172–202.

[180] Torbjörn Granlund, Richard Kenner, Eliminating branches using a superoptimizer and the GNU C compiler, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 27 (7) (July 1992) 341–352.

[181] David Gries, Compiler Construction for Digital Computers, John Wiley and Sons, New York, NY, 1971.

[182] Daniel Grove, Linda Torczon, Interprocedural constant propagation: A study of jump function implementations, in: Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 28 (6) (June 1993) 90–99.

[183] Rajiv Gupta, Mary Lou Soffa, Region scheduling: An approach for detecting and redistributing parallelism, IEEE Transactions on Software Engineering SE-16 (4) (April 1990) 421–431.

[184] Rajiv Gupta, Mary Lou Soffa, Tim Steele, Register allocation via clique separators, in: Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 24 (7) (July 1989) 264–274.

[185] Sebastian Hack, Register Allocation for Programs in SSA Form, PhD thesis, Universität Karlsruhe, Karlsruhe, Germany, October 2006.

[186] Sebastian Hack, Gerhard Goos, Optimal register allocation for SSA-form programs in polynomial time, Information Processing Letters 98 (4) (May 2006) 150–155.

[187] Max Hailperin, Cost-optimal code motion, ACM Transactions on Programming Languages and Systems (TOPLAS) 20 (6) (November 1998) 1297–1322.

[188] Gilbert Joseph Hansen, Adaptive Systems for the Dynamic Run-Time Optimization of Programs, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1974.

[189] David R. Hanson, Fast allocation and deallocation of memory based on object lifetimes, Software—Practice and Experience 20 (1) (January 1990) 5–12.

[190] Dov Harel, A linear time algorithm for finding dominators in flow graphs and related problems, in: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC), Providence, RI, May 1985, pp. 185–194.

[191] William H. Harrison, A class of register allocation algorithms, Technical Report RC-5342, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1975.

[192] William H. Harrison, A new strategy for code generation—The general purpose optimizing compiler, IEEE Transactions on Software Engineering SE-5 (4) (July 1979) 367–373.

[193] Timothy J. Harvey, Reducing the Impact of Spill Code, MS thesis, Department of Computer Science, Rice University, Houston, TX, May 1998.

[194] Amir H. Hashemi, David R. Kaeli, Brad Calder, Efficient procedure mapping using cache line coloring, in: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 171–182.

[195] Philip J. Hatcher, Thomas W. Christopher, High-quality code generation via bottom-up tree pattern matching, in: Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, FL, January 1986, pp. 119–130.

[196] Matthew S. Hecht, Jeffrey D. Ullman, Characterizations of reducible flow graphs, Journal of the ACM 21 (3) (July 1974) 367–375.

[197] Matthew S. Hecht, Jeffrey D. Ullman, A simple algorithm for global data flow analysis problems, SIAM Journal on Computing 4 (4) (1975) 519–532.

[198] J. Heller, Sequencing aspects of multiprogramming, Journal of the ACM 8 (3) (July 1961) 426–439.

[199] John L. Hennessy, Thomas Gross, Postpass code optimization of pipeline constraints, ACM Transactions on Programming Languages and Systems (TOPLAS) 5 (3) (July 1983) 422–448.

[200] John L. Hennessy, David A. Patterson, Computer Architecture, Sixth Edition: A Quantitative Approach, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2017.

[201] Vincent P. Heuring, The automatic generation of fast lexical analysers, Software—Practice and Experience 16 (9) (September 1986) 801–808.

[202] Michael Hind, Michael Burke, Paul Carini, Jong-Deok Choi, Interprocedural pointer alias analysis, ACM Transactions on Programming Languages and Systems (TOPLAS) 21 (4) (July 1999) 848–894.

[203] Michael Hind, Anthony Pioli, Which pointer analysis should I use?, in: Proceedings of the International Symposium on Software Testing and Analysis, ACM SIGSOFT Software Engineering Notes 25 (5) (September 2000) 113–123.

[204] Christoph M. Hoffmann, Michael J. O'Donnell, Pattern matching in trees, Journal of the ACM 29 (1) (January 1982) 68–95.

[205] John E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Zvi Kohavi, Azaria Paz (Eds.), Theory of Machines and Computations: Proceedings, Academic Press, New York, NY, 1971, pp. 189–196.

[206] John E. Hopcroft, Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.

[207] Ellis Horowitz, Sartaj Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Inc., Potomac, MD, 1978.

[208] Lawrence P. Horwitz, Richard M. Karp, Raymond E. Miller, Shmuel Winograd, Index register allocation, Journal of the ACM 13 (1) (January 1966) 43–61.

[209] Susan Horwitz, Phil Pfeiffer, Thomas Reps, Dependence analysis for pointer variables, in: Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 24 (7) (July 1989) 28–40.

[210] Brett L. Huber, Path-selection heuristics for dominator-path scheduling, Master's thesis, Computer Science Department, Michigan Technological University, Houghton, MI, October 1995.

[211] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, Daniel M. Lavery, The superblock: An effective technique for VLIW and superscalar compilation, Journal of Supercomputing—Special Issue on Instruction Level Parallelism 7 (1–2) (May 1993) 229–248.

[212] ISO IEC, Programming Languages—C++, Standard 14882:2017, International Organization for Standardization, Geneva, Switzerland, 2017.

[213] Peter Zilahy Ingerman, Thunks: A way of compiling procedure statements with some comments on procedure declarations, Communications of the ACM 4 (1) (January 1961) 55–58.

[214] Edgar T. Irons, A syntax directed compiler for Algol 60, Communications of the ACM 4 (1) (January 1961) 51–55.

[215] Mark Scott Johnson, Terrence C. Miller, Effectiveness of a machine-level, global optimizer, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 99–108.

[216] Stephen C. Johnson, Yacc: Yet another compiler-compiler, Technical Report 32 (Computing Science), AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[217] Stephen C. Johnson, A tour through the portable C compiler, in: Unix Programmer's Manual, 7th Edition, vol. 2b, AT&T Bell Laboratories, Murray Hill, NJ, January 1979.

[218] Walter L. Johnson, James H. Porter, Stephanie I. Ackley, Douglas T. Ross, Automatic generation of efficient lexical processors using finite state techniques, Communications of the ACM 11 (12) (December 1968) 805–813.

[219] Douglas W. Jones, How (not) to code a finite state machine, ACM SIGPLAN Notices 23 (8) (August 1988) 19–22.

[220] S.M. Joshi, Dhananjay M. Dhamdhere, A composite hoisting-strength reduction transformation for global program optimization, International Journal of Computer Mathematics 11 (1) (1982) 21–44 (part I), International Journal of Computer Mathematics 11 (2) (1982) 111–126 (part II).

[221] John B. Kam, Jeffrey D. Ullman, Global data flow analysis and iterative algorithms, Journal of the ACM 23 (1) (January 1976) 158–171.

[222] John B. Kam, Jeffrey D. Ullman, Monotone data flow analysis frameworks, Acta Informatica 7 (3) (September 1977) 305–317.

[223] Tadao Kasami, An efficient recognition and syntax analysis algorithm for context-free languages, Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.

[224] Ken Kennedy, A global flow analysis algorithm, International Journal of Computer Mathematics 3 (Section A) (December 1971) 5–15.

[225] Ken Kennedy, Global Flow Analysis and Register Allocation for Simple Code Structures, PhD thesis, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1971.

[226] Ken Kennedy, Global dead computation elimination, SETL Newsletter 111, Courant Institute of Mathematical Sciences, New York University, New York, NY, August 1973.

[227] Ken Kennedy, Reduction in strength using hashed temporaries, in: SETL Newsletter, vol. 102, Courant Institute of Mathematical Sciences, New York University, New York, NY, March 1973.

[228] Ken Kennedy, Use-definition chains with applications, Computer Languages 3 (3) (1978) 163–179.

[229] Ken Kennedy, A survey of data flow analysis techniques, in: Neil D. Jones, Steven S. Muchnik (Eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[230] Ken Kennedy, Linda Zucconi, Applications of graph grammar for program control flow analysis, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, CA, January 1977, pp. 72–85.

[231] Robert Kennedy, Fred C. Chow, Peter Dahl, Shin-Ming Liu, Raymond Lo, Mark Streich, Strength reduction via SSAPRE, in: Proceedings of the Seventh International Conference on Compiler Construction (CC '98), in: Lecture Notes in Com-

puter Science (LNCS), vol. 1383, Springer-Verlag, Heidelberg, Germany, March 1998, pp. 144–158.

[232] Daniel R. Kerns, Susan J. Eggers, Balanced scheduling: Instruction scheduling when memory latency is uncertain, in: Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 28 (6) (June 1993) 278–289.

[233] Robert R. Kessler, Peep—An architectural description driven peephole optimizer, in: Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 106–110.

[234] Gary A. Kildall, A unified approach to global program optimization, in: Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, MA, October 1973, pp. 194–206.

[235] Stephen C. Kleene, Representation of events in nerve nets and finite automata, in: Claude E. Shannon, John McCarthy (Eds.), Automata Studies, in: Annals of Mathematics Studies, vol. 34, Princeton University Press, Princeton, NJ, 1956, pp. 3–41.

[236] Jens Knoop, Oliver Rüthing, Bernhard Steffen, Lazy code motion, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 27 (7) (July 1992) 224–234.

[237] Jens Knoop, Oliver Rüthing, Bernhard Steffen, Lazy strength reduction, International Journal of Programming Languages 1 (1) (March 1993) 71–91.

[238] Donald E. Knuth, A history of writing compilers, Computers and Automation 11 (12) (December 1962) 8–18, Reprinted in: Bary W. Pollack (Ed.), Compiler Techniques, Auerbach, Princeton, NJ, 1972, pp. 38–56.

[239] Donald E. Knuth, On the translation of languages from left to right, Information and Control 8 (6) (December 1965) 607–639.

[240] Donald E. Knuth, The Art of Computer Programming, Addison-Wesley, Reading, MA, 1973.

[241] Dexter C. Kozen, Automata and Computability, Springer-Verlag, New York, NY, 1997.

[242] Glenn Krasner (Ed.), Smalltalk-80: Bits of History, Words of Advice, Addison-Wesley, Reading, MA, August 1983.

[243] Sanjay M. Krishnamurthy, A brief survey of papers on scheduling for pipelined processors, SIGPLAN Notices 25 (7) (July 1990) 97–106.

[244] Steven M. Kurlander, Charles N. Fischer, Zero-cost range splitting, in: Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 29 (6) (June 1994) 257–265.

[245] Monica Lam, Software pipelining: An effective scheduling technique for VLIW machines, in: Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23 (7) (July 1988) 318–328.

[246] David Alex Lamb, Construction of a peephole optimizer, Software—Practice and Experience 11 (6) (June 1981) 639–647.

[247] William Landi, Barbara G. Ryder, Pointer-induced aliasing: A problem taxonomy, in: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, FL, January 1991, pp. 93–103.

[248] David Landskov, Scott Davidson, Bruce Shriver, Patrick W. Mallett, Local microcode compaction techniques, ACM Computing Surveys 12 (3) (September 1980) 261–294.

[249] Rudolf Landwehr, Hans-Stephan Jansohn, Gerhard Goos, Experience with an automatic code generator generator, in: Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, ACM SIGPLAN Notices 17 (6) (June 1982) 56–66.

[250] S.S. Lavrov, Store economy in closed operator schemes, Journal of Computational Mathematics and Mathematical Physics 1 (4) (1961) 687–701, English translation in U.S.S.R. Computational Mathematics and Mathematical Physics 3 (1962) 810–828.

[251] Vincent Lefévre, Multiplication by an integer constant, Research Report 4192, INRIA, France, May 2001.

[252] Thomas Lengauer, Robert Endre Tarjan, A fast algorithm for finding dominators in a flowgraph, ACM Transactions on Programming Languages and Systems (TOPLAS) 1 (1) (July 1979) 121–141.

[253] Philip M. Lewis, Richard E. Stearns, Syntax-directed transduction, Journal of the ACM 15 (3) (July 1968) 465–488.

[254] Vincenzo Liberatore, Martin Farach-Colton, Ulrich Kremer, Evaluation of algorithms for local register allocation, in: Proceedings of the Eighth International Conference on Compiler Construction (CC '99), in: Lecture Notes in Computer Science (LNCS), vol. 1575, Springer-Verlag, Heidelberg, Germany, 1999, pp. 137–152.

[255] Henry Lieberman, Carl Hewitt, A real-time garbage collector based on the lifetimes of objects, Communications of the ACM 26 (6) (June 1983) 419–429.

[256] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, Alan Snyder, CLU Reference Manual, Lecture Notes in Computer Science (LNCS), vol. 114, Springer-Verlag, Heidelberg, Germany, 1981.

[257] Jack L. Lo, Susan J. Eggers, Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism, in: Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 30 (6) (June 1995) 151–162.

[258] Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, Peng Tu, Register promotion by sparse partial redundancy elimination of loads and stores, in: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 33 (5) (May 1998) 26–37.

[259] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W.D. Lichtenstein, Robert P. Nix, John S. O'Donnell, John C. Ruttenburg, The multiflow trace scheduling compiler, The Journal of Supercomputing—Special Issue 7 (1–2) (March 1993) 51–142.

[260] Edward S. Lowry, C.W. Medlock, Object code optimization, Communications of the ACM 12 (1) (January 1969) 13–22.

[261] John Lu, Keith D. Cooper, Register promotion in C programs, in: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 308–319.

[262] John Lu, Rob Shillingsburg, Clean: Removing useless control flow, Unpublished, Department of Computer Science, Rice University, Houston, TX, June 1994.

[263] Peter Lucas, Die Strukturanalyse von Formelübersetzern, Elektronische Rechenanlagen 3 (4) (August 1961) 159–167.

[264] Peter W. Markstein, Victoria Markstein, F. Kenneth Zadeck, Reassociation and strength reduction, Unpublished book chapter.

[265] Henry Massalin, Superoptimizer—A look at the smallest program, in: Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II), ACM SIGPLAN Notices 22 (10) (October 1987) 122–126.

[266] John McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, Communications of the ACM 3 (4) (April 1960) 184–195.

[267] John McCarthy, Lisp—Notes on its past and future, in: Proceedings of the 1980 ACM Conference on Lisp and Functional Programming, Stanford University, Stanford, CA, 1980, pp. v–viii.

[268] William M. McKeeman, Peephole optimization, Communications of the ACM 8 (7) (July 1965) 443–444.

[269] Kathryn S. McKinley, Steve Carr, Chau-Wen Tseng, Improving data locality with loop transformations, ACM Transactions on Programming Languages and Systems (TOPLAS) 18 (4) (July 1996) 424–453.

[270] Robert McNaughton, H. Yamada, Regular expressions and state graphs for automata, IRE Transactions on Electronic Computers EC-9 (1) (March 1960) 39–47.

[271] Robert Metzger, Sean Stroud, Interprocedural constant propagation: An empirical study, ACM Letters on Programming Languages and Systems (LOPLAS) 2 (1–4) (March–December 1993) 213–232.

[272] Ana Milanova, Atanas Rountev, Barbara G. Ryder, Precise call graphs for C programs with function pointers, Automated Software Engineering 11 (1) (January 2004) 7–26.

[273] Terrence C. Miller, Tentative Compilation: A Design for an APL Compiler, PhD thesis, Yale University, New Haven, CT, May 1978. See also the paper of the same title in: Proceedings of the International Conference on APL: Part 1, New York, NY, 1979, pp. 88–95.

[274] Robin Milner, Mads Tofte, Robert Harper, David MacQueen, The Definition of Standard ML—Revised, MIT Press, Cambridge, MA, 1997.

[275] James Strother Moore, The Interlisp Virtual Machine specification, Technical Report CSL 76-5, Xerox Palo Alto Research Center, Palo Alto, CA, September 1976.

[276] Etienne Morel, Claude Renvoise, Global optimization by suppression of partial redundancies, Communications of the ACM 22 (2) (February 1979) 96–103.

[277] Robert Morgan, Building an Optimizing Compiler, Digital Press (an imprint of Butterworth–Heineman), Boston, MA, February 1998.

[278] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, Salem Reyen, Combining register allocation and instruction scheduling, Technical Report 698, Courant Institute of Mathematical Sciences, New York University, New York, NY, July 1995.

[279] Steven S. Muchnick, Advanced Compiler Design & Implementation, Morgan Kaufmann, San Francisco, CA, 1997.

[280] Frank Mueller, David B. Whalley, Avoiding unconditional jumps by code replication, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 27 (7) (July 1992) 322–330.

[281] Thomas P. Murtagh, An improved storage management scheme for block structured languages, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (3) (July 1991) 372–398.

[282] Peter Naur (editor), J.W. Backus, F.L. Bauer, J. Green, C. Katz, J. McCarthy, A.J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J.H. Wegstein, A. van Wijngaarden, M. Woodger, Revised report on the algorithmic language Algol 60, Communications of the ACM 6 (1) (January 1963) 1–17.

[283] E. Ketcha Ngassam, Bruce W. Watson, Derrick G. Kourie, Hardcoding finite state automata processing, in: Proceedings of the 2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology (SAICSIT '03), September 2003, pp. 111–121.

[284] Brian R. Nickerson, Graph coloring register allocation for processors with multi-register operands, in: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 25 (6) (June 1990) 40–52.

[285] Cindy Norris, Lori L. Pollock, A scheduler-sensitive global register allocator, in: Proceedings of Supercompting '93, Portland, OR, November 1993, pp. 804–813.

[286] Cindy Norris, Lori L. Pollock, An experimental study of several cooperative register allocation and instruction scheduling strategies, in: Proceedings of the Twenty-Eighth Annual International Symposium on Microarchitecture (MICRO-28), Ann Arbor, MI, December 1995, pp. 169–179.

[287] Kristen Nygaard, Ole-Johan Dahl, The development of the SIMULA languages, in: Proceedings of the First ACM SIGPLAN Conference on the History of Programming Languages, Los Angeles, CA, USA, ACM SIGPLAN Notices 13 (8) (August 1978) 245–272.

[288] Michael Paleczny, Christopher A. Vick, Cliff Click, The Java HotSpot Server Compiler, in: JVM '01: Proceedings of the 2001 Java Virtual Machine Research and Technology Symposium, vol. 1, Monterey, CA, April 2001, pp. 1–12.

[289] Jinpyo Park, Soo-Mook Moon, Optimistic register coalescing, in: Proceedings of the 1998 International Conference on Parallel Architecture and Compilation Techniques (PACT), October 1998, pp. 196–204.

[290] Eduardo Pelegrí-Llopart, Susan L. Graham, Optimal code generation for expression trees: An application of BURS theory, in: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, CA, January 1988, pp. 294–308.

[291] Thomas J. Pennello, Very fast LR parsing, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 145–151.

[292] Fernando Magno Quintão Pereira, Jens Palsberg, Register allocation via coloring of chordal graphs, in: Proceedings of the Asian Symposium on Programming Languages and Systems (ASPLAS '05), in: Lecture Notes in Computer Science (LNCS), vol. 3780, Springer, Berlin, Heidelberg, November 2005, pp. 315–329.

[293] Karl Pettis, Robert C. Hansen, Profile guided code positioning, in: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 25 (6) (June 1990) 16–27.

[294] Shlomit S. Pinter, Register allocation with instruction scheduling: A new approach, in: Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 28 (6) (June 1993) 248–257.

[295] Gordon D. Plotkin, Call-by-name, call-by-value and the λ-calculus, Theoretical Computer Science 1 (2) (December 1975) 125–159.

[296] Massimiliano Poletto, Vivek Sarkar, Linear scan register allocation, ACM Transactions on Programming Languages and Systems (TOPLAS) 21 (5) (September 1999) 895–913.

[297] Todd A. Proebsting, Simple and efficient BURS table generation, in: Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 27 (7) (July 1992) 331–340.

[298] Todd A. Proebsting, Optimizing an ANSI C interpreter with superoperators, in: Proceedings of the Twenty-Second ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA, January 1995, pp. 322–332.

[299] Todd A. Proebsting, Charles N. Fischer, Linear-time, optimal code scheduling for delayed-load architectures, in: Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 26 (6) (June 1991) 256–267.

[300] Reese T. Prosser, Applications of boolean matrices to the analysis of flow diagrams, in: Proceedings of the Eastern Joint Computer Conference, Institute of Radio Engineers, New York, NY, December 1959, pp. 133–138.

[301] Paul W. Purdom Jr., Edward F. Moore, Immediate predominators in a directed graph [H], Communications of the ACM 15 (8) (August 1972) 777–778.

[302] Michael O. Rabin, Dana Scott, Finite automata and their decision problems, IBM Journal of Research and Development 3 (2) (April 1959) 114–125.

[303] Brian Randell, L.J. Russell, Algol 60 Implementation: The Translation and Use of Algol 60 Programs on a Computer, Academic Press, London, UK, January 1964.

[304] Bob R. Rau, C.D. Glaeser, Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing, in: Proceedings of the Fourteenth Annual Workshop on Microprogramming (MICRO-14), Chatham, MA, December 1981, pp. 183–198.

[305] John H. Reif, Symbolic programming analysis in almost linear time, in: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, AZ, January 1978, pp. 76–83.

[306] John H. Reif, Harry R. Lewis, Symbolic evaluation and the global value graph, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, CA, January 1977, pp. 104–118.

[307] Thomas Reps, "Maximal-munch" tokenization in linear time, ACM Transactions on Programming Languages and Systems (TOPLAS) 20 (2) (March 1998) 259–273.

[308] Martin Richards, The portability of the BCPL compiler, Software—Practice and Experience 1 (2) (April–June 1971) 135–146.

[309] Steve Richardson, Mahadevan Ganapathi, Interprocedural analysis versus procedure integration, Information Processing Letters 32 (3) (August 1989) 137–142.

[310] Ronald Rivest, On self-organizing sequential search heuristics, Communications of the ACM 19 (2) (February 1976) 63–67.

[311] Anne Rogers, Kai Li, Software support for speculative loads, in: Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), ACM SIGPLAN Notices 27 (9) (September 1992) 38–50.

[312] Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck, Global value numbers and redundant computations, in: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, CA, January 1988, pp. 12–27.

[313] Daniel J. Rosenkrantz, Richard Edwin Stearns, Properties of deterministic top-down grammars, Information and Control 17 (3) (October 1970) 226–256.

[314] Barbara G. Ryder, Constructing the call graph of a program, IEEE Transactions on Software Engineering SE-5 (3) (May 1979) 216–226.

[315] A.V.S. Sastry, Roy D.C. Ju, A new algorithm for scalar register promotion based on SSA form, in: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 33 (5) (May 1998) 15–25.

[316] Kirk Sattley, Allocation of storage for arrays in Algol 60, Communications of the ACM 4 (1) (January 1961) 60–65.

[317] Randolph G. Scarborough, Harwood G. Kolsky, Improved optimization of FORTRAN object programs, IBM Journal of Research and Development 24 (6) (November 1980) 660–676.

[318] Philip J. Schielke, Stochastic Instruction Scheduling, PhD thesis, Department of Computer Science, Rice University, Houston, TX, May 2000, Technical Report TR00-370, Computer Science Department, Rice University, 2000.

[319] Herb Schorr, William M. Waite, An efficient machine-independent procedure for garbage collection in various list structures, Communications of the ACM 10 (8) (August 1967) 501–506.

[320] Jacob T. Schwartz, On programming: An interim report on the SETL project. Installment II: The SETL language and examples of its use, Technical Report, Courant Institute of Mathematical Sciences, New York University, New York, NY, October 1973.

[321] Ravi Sethi, Jeffrey D. Ullman, The generation of optimal code for arithmetic expressions, Journal of the ACM 17 (4) (October 1970) 715–728.

[322] Marc Shapiro, Susan Horwitz, Fast and accurate flow-insensitive points-to analysis, in: Proceedings of the Twenty-Fourth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Paris, France, January 1997, pp. 1–14.

[323] Robert M. Shapiro, Harry Saint, The representation of algorithms, Technical Report CA-7002-1432, Massachusetts Computer Associates, February 1970.

[324] Peter B. Sheridan, The arithmetic translator-compiler of the IBM FORTRAN automatic coding system, Communications of the ACM 2 (2) (February 1959) 9–21.

[325] Olin Shivers, Control flow analysis in Scheme, in: Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 23 (7) (July 1988) 164–174.

[326] L. Taylor Simpson, Value-Driven Redundancy Elimination, PhD thesis, Department of Computer Science, Rice University, Houston, TX, 1996, Technical Report TR96-308, Computer Science Department, Rice University, 1996.

[327] Michael Sipser, Introduction to the Theory of Computation, PWS Publishing Co., Boston, MA, December 1996.

[328] Richard L. Sites, Daniel R. Perkins, Universal P-code definition, version 0.2, Technical Report 78-CS-C29, Department of Applied Physics and Information Sciences, University of California, San Diego, San Diego, CA, January 1979.

[329] Daniel Dominic Sleator, Robert Endre Tarjan, Amortized efficiency of list update and paging rules, Communications of the ACM 28 (2) (February 1985) 202–208.

[330] Michael D. Smith, Mark Horowitz, Monica S. Lam, Efficient superscalar performance through boosting, in: Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), ACM SIGPLAN Notices 27 (9) (September 1992) 248–259.

[331] Michael D. Smith, Norman Ramsey, Glenn Holloway, A generalized algorithm for graph-coloring register allocation, in: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 39 (6) (May 2004) 277–288.

[332] Mark Smotherman, Sanjay M. Krishnamurthy, P.S. Aravind, David Hunnicutt, Efficient DAG construction and heuristic calculation for instruction scheduling, in: Proceedings of the Twenty-Fourth Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-24), Albuquerque, NM, August 1991, pp. 93–102.

[333] Arthur Sorkin, Some comments on "A solution to a problem with Morel and Renvoise's 'Global optimization by suppression of partial redundancies,"' ACM Transactions on Programming Languages and Systems (TOPLAS) 11 (4) (October 1989) 666–668.

[334] Thomas C. Spillman, Exposing side-effects in a PL/1 optimizing compiler, in: Foundations and Systems, in: Information Processing: Proceedings of IFIP Congress 1971, vol. 1, Ljubljana, Yugoslavia, August 1971, North-Holland Publishing Company, Amsterdam, 1972, pp. 376–381.

[335] Guy L. Steele Jr., Rabbit: A compiler for Scheme, Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, May 1978.

[336] Guy L. Steele Jr., Richard P. Gabriel, The evolution of LISP, in: Thomas J. Bergin, Richard G. Gibson (Eds.), History of Programming Languages—II, ACM Press, New York, NY, January 1996, pp. 233–330.

[337] Mark Stephenson, Saman Amarasinghe, Predicting unroll factors using supervised classification, in: CGO '05: Proceedings of the International Symposium on Code Generation and Optimization, San Jose, CA, IEEE Computer Society, Washington, DC, March 2005, pp. 123–134.

[338] Philip H. Sweany, Steven J. Beaty, Post-compaction register assignment in a retargetable compiler, in: Proceedings of the Twenty-Third Annual International Symposium and Workshop on Microprogramming and Microarchitecture (MICRO-23), Orlando, FL, November 1990, pp. 107–116.

[339] Philip H. Sweany, Steven J. Beaty, Dominator-path scheduling—A global scheduling method, in: Proceedings of the Twenty-Fifth Annual International Symposium on Microarchitecture (MICRO-25), ACM SIGMICRO Newsletter 23 (1–2) (December 1992) 260–263.

[340] Deian Tabakov, Moshe Y. Vardi, Experimental evaluation of classical automata constructions, in: Proceedings of the Twelfth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '05), in: Lecture Notes in Computer Science (LNCS), vol. 3835, Springer, Berlin, Heidelberg, December 2005, pp. 396–411.

[341] Robert Endre Tarjan, Testing flow graph reducibility, Journal of Computer and System Sciences 9 (3) (December 1974) 355–365.

[342] Robert Endre Tarjan, Fast algorithms for solving path problems, Journal of the ACM 28 (3) (July 1981) 594–614.

[343] Robert Endre Tarjan, A unified approach to path problems, Journal of the ACM 28 (3) (July 1981) 577–593.

[344] Robert Endre Tarjan, John H. Reif, Symbolic program analysis in almost-linear time, SIAM Journal on Computing 11 (1) (February 1982) 81–93.

[345] Ken Thompson, Programming techniques: Regular expression search algorithm, Communications of the ACM 11 (6) (1968) 419–422.

[346] Steven W.K. Tjiang, Twig reference manual, Technical Report CSTR 120, Computing Sciences, AT&T Bell Laboratories, Murray Hill, NJ, January 1986.

[347] Linda Torczon, Compilation Dependences in an Ambitious Optimizing Compiler, PhD thesis, Department of Computer Science, Rice University, Houston, TX, 1985.

[348] Jeffrey D. Ullman, Fast algorithms for the elimination of common subexpressions, Acta Informatica 2 (3) (July 1973) 191–213.

[349] David Ungar, Generation scavenging: A non-disruptive high performance storage reclamation algorithm, in: Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGSOFT Software Engineering Notes 9 (3) (May 1984) 157–167.

[350] Anand Venkat, Mary Hall, Michelle Strout, Loop and data transformations for sparse matrix code, in: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 50 (6) (June 2015) 521–532.

[351] Victor Vyssotsky, Peter Wegner, A graph theoretical FORTRAN source language analyzer, Manuscript, AT&T Bell Laboratories, Murray Hill, NJ, 1963.

[352] William Waite, Gerhard Goos, Compiler Construction, Springer-Verlag, New York, NY, 1984.

[353] William M. Waite, The cost of lexical analysis, Software—Practice and Experience 16 (5) (May 1986) 473–488.

[354] David W. Wall, Global register allocation at link time, in: Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices 21 (7) (July 1986) 264–275.

[355] Bruce W. Watson, A taxonomy of deterministic finite automata minimization algorithms, Computing Science Report 93/44, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, December 1993.

[356] Bruce W. Watson, A fast new semi-incremental algorithm for the construction of minimal acyclic DFAs, in: Proceedings of the Third International Workshop on Implementing Automata (WIA 1998), in: Lecture Notes in Computer Science (LNCS), vol. 1660, Springer, Berlin, Heidelberg, 1999, pp. 121–132.

[357] Bruce W. Watson, A fast and simple algorithm for constructing minimal acyclic deterministic finite automata, Journal of Universal Computer Science 8 (2) (February 2002) 363–367.

[358] Mark N. Wegman, F. Kenneth Zadeck, Constant propagation with conditional branches, in: Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, LA, January 1985, pp. 291–299.

[359] Mark N. Wegman, F. Kenneth Zadeck, Constant propagation with conditional branches, ACM Transactions on Programming Languages and Systems (TOPLAS) 13 (2) (April 1991) 181–210.

[360] William E. Weihl, Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables, in: Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, NV, January 1980, pp. 83–94.

[361] Clark Wiedmann, Steps toward an APL compiler, in: Proceedings of the International Conference on APL, ACM SIGAPL APL Quote Quad 9 (4) (June 1979) 321–328.

[362] Paul R. Wilson, Uniprocessor garbage collection techniques, in: Proceedings of the International Workshop on Memory Management, in: Lecture Notes in Computer Science (LNCS), vol. 637, Springer-Verlag, Heidelberg, Germany, 1992, pp. 1–42.

[363] Robert P. Wilson, Monica S. Lam, Efficient context-sensitive pointer analysis for C programs, in: Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 30 (6) (June 1995) 1–12.

[364] Michael Wolfe, High Performance Compilers for Parallel Computing, Addison Wesley, Redwood City, CA, 1996.

[365] Derick Wood, The theory of left-factored languages, part 1, The Computer Journal 12 (4) (November 1969) 349–356.

[366] Derick Wood, The theory of left-factored languages, part 2, The Computer Journal 13 (1) (February 1970) 55–62.

[367] Derick Wood, A further note on top-down deterministic languages, The Computer Journal 14 (4) (November 1971) 396–403.

[368] William Wulf, Richard K. Johnsson, Charles B. Weinstock, Steven O. Hobbs, Charles M. Geschke, The Design of an Optimizing Compiler, Programming Languages Series, American Elsevier Publishing Company, Inc., New York, NY, 1975.

[369] Cliff Young, David S. Johnson, David R. Karger, Michael D. Smith, Near-optimal intraprocedural branch alignment, in: Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices 32 (5) (May 1997) 183–193.

[370] Daniel H. Younger, Recognition and parsing of context-free languages in time n^3, Information and Control 10 (2) (1967) 189–208.

[371] F. Kenneth Zadeck, Incremental data flow analysis in a structured program editor, in: Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19 (6) (June 1984) 132–143.

[372] Weilei Zhang, Barbara G. Ryder, Automatic construction of accurate application call graph with library call abstraction for Java, Journal of Software Maintenance and Evolution: Research and Practice 19 (4) (July 2007) 231–252.

This page intentionally left blank

Index

This page intentionally left blank

This page intentionally left blank

ENGINEERING A COMPILER

Keith D. Cooper & Linda Torczon

THIRD EDITION

The classic introduction to compiler construction is now fully updated with new techniques and a more practical focus!

This new edition of ***Engineering a Compiler*** is full of technical updates, new material covering the latest developments in compiler technology, and a fundamental change in the presentation of the middle section of the book, with new chapters focusing on s*emantic elaboration* (the problems that arise in generating code from the *ad-hoc syntax-directed translation* schemes in a generated parser), on runtime support for naming and addressability, and on code shape for expressions, assignments, and control-structures. Leading educators and researchers Keith Cooper and Linda Torczon have revised their popular text with a fresh approach to learning important techniques for constructing a modern compiler, combining basic principles with pragmatic insights from their own experience building state-of-the-art compilers.

Key features of this edition:

- In-depth treatment of algorithms and techniques used in the front end of a modern compiler
- Focus on code optimization and code generation, the primary areas of recent research and development
- New chapters in the middle section of the book, focusing on how compilers (and interpreters) implement abstraction, tying the underlying knowledge to students' own experience and to the languages in which they have been taught to program
- Chapter 13 on register allocation, fully rewritten to focus on bottom-up methods of register allocation at the local scope
- Chapter 14 providing an overview of runtime optimization, the challenges that arise in building a just-in-time (JIT) compiler, and some historical perspective on JIT compilers from the 1960s to present day

Cooper and Torczon's ***Engineering a Compiler***, unlike many other compiler texts, focuses equally on all aspects of compiler construction, from the design of a compiler front end and the design and construction of tools to build front ends, to the mapping of source code into the compiler's intermediate form, to the subject of code optimization, and finally to the algorithms used in the compiler's back end.

About the Authors

Keith D. Cooper is the Doerr Professor in Computational Engineering at Rice University. He has served as Chair of Rice's Computer Science Department, Chair of its Computational and Applied Mathematics Department, and Associate Dean for Research in Rice's Engineering School. He is a Fellow of the ACM.

Linda Torczon had a long career as a Senior Research Scientist in Rice University's Computer Science Department. Dr. Torczon served as Executive Director of the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center. She also served as the Executive Director of HiPerSoft, of the Los Alamos Computer Science Institute, and of the Virtual Grid Application Development Software Project. She was a principal investigator on the DARPA-sponsored Platform Aware Compilation Environment project, and she also served as the director of Rice's Computer Science Professional Master's Degree program.

ISBN 978-0-12-815412-0

9 780128 154120

ELSEVIER

MK
MORGAN KAUFMANN PUBLISHERS
An imprint of Elsevier
elsevier.com/books-and-journals