Preface

Vision

Compiler construction brings together techniques from disparate parts of Computer Science. The compiler deals with many big-picture issues. At its simplest, a compiler is just a computer program that takes as input one potentially executable program and produces as output another, related, potentially executable program. As part of this translation, the compiler must perform syntax analysis to determine if the input program is valid. To map that input program onto the finite resources of a target computer, the compiler must manipulate several distinct name spaces, allocate several different kinds of resources, and synchronize the behavior of different run-time components. For the output program to have reasonable performance, it must manage hardware latencies in functional units, predict the flow of execution and the demand for memory, and reason about the independence and dependence of different machine-level operations in the program.

Open up a compiler and you are likely to find greedy heuristic searches that explore large solution spaces, finite automata that recognize words in the input, fixed-point algorithms that help reason about program behavior, simple theorem provers and algebraic simplifiers that try to predict the values of expressions, pattern-matchers for both strings and trees that match abstract computations to machine-level operations, solvers for diophantine equations and Pressburger arithmetic used to analyze array subscripts, and techniques such as hash tables, graph algorithms, and sparse set implementations used in myriad applications,

The lore of compiler construction includes both amazing success stories about the application of theory to practice and humbling stories about the limits of what we can do. On the success side, modern scanners are built by applying the theory of regular languages to automatic construction of recognizers. Lr parsers use the same techniques to perform the handle-recognition that drives a shift-reduce parser. Data-flow analysis (and its cousins) apply lattice theory to the analysis of programs in ways that are both useful and clever. Some of the problems that a compiler faces are truly hard; many clever approximations and heuristics have been developed to attack these problems.

On the other side, we have discovered that some of the problems that compilers must solve are quite hard. For example, the back end of a compiler for a modern superscalar machine must approximate the solution to two or more interacting NP-complete problems (instruction scheduling, register allocation, and, perhaps, instruction and data placement). These NP-complete problems, however, look easy next to problems such as algebraic reassociation of expressions. This problem admits a huge number of solutions; to make matters worse, the desired solution is somehow a function of the other transformations being applied in the compiler. While the compiler attempts to solve these problems (or approximate their solution), we constrain it to run in a reasonable amount of time and to consume a modest amount of space. Thus, a good compiler for a modern superscalar machine is an artful blend of theory, of practical knowledge, of engineering, and of experience.

This text attempts to convey both the art and the science of compiler construction. We have tried to cover a broad enough selection of material to show the reader that real tradeoffs exist, and that the impact of those choices can be both subtle and far-reaching. We have limited the material to a manageable amount by omitting techniques that have become less interesting due to changes in the marketplace, in the technology of languages and compilers, or in the availability of tools. We have replaced this material with a selection of subjects that have direct and obvious importance today, such as instruction scheduling, global register allocation, implementation object-oriented languages, and some introductory material on analysis and transformation of programs.

Target Audience

The book is intended for use in a first course on the design and implementation of compilers. Our goal is to lay out the set of problems that face compiler writers and to explore some of the solutions that can be used to solve these problems. The book is not encyclopedic; a reader searching for a treatise on *Earley's algorithm* or *left-corner parsing* may need to look elsewhere. Instead, the book presents a pragmatic selection of practical techniques that you might use to build a modern compiler.

Compiler construction is an exercise in engineering design. The compiler writer must choose a path through a decision 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 of way.

Thus, there is no *right* answer for these problems. 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 (IR) for the compiler has a profound impact on the rest of the compiler, from space and time requirements through the ease with which different algorithms can be applied. The decision, however, is given short shrift in most books (and papers). Chapter 6 examines the space of IRs and some of the issues that should be considered in selecting an IR. We raise the issue again at many points in the book—both directly in the text and indirectly in the questions at the end of each chapter.

This book tries to explore the design space – to present some of the ways problems have been solved and the constraints that made each of those solutions attractive at the time. By understanding the parameters of the problem and their impact on compiler design, we hope to convey both the breadth of possibility and the depth of the problems.

This book departs from some of the accepted conventions for compiler construction textbooks. For example, we use several different programming languages in the examples. It makes little sense to describe call-by-name parameter passing in C, so we use Algol-60. It makes little sense to describe tail-recursion in Fortran, so we use Scheme. This multi-lingual approach is realistic; over the course of the reader's career, the "language of the future" will change several times. (In the past thirty years, Algol-68, APL, PL/I, Smalltalk, C, Modula-3, C++, and even ADA have progressed from being "the language of the future" to being the "language of the future of the past.") Rather than provide ten to twenty homework-level questions at the end of each chapter, we present a couple of questions suitable for a mid-term or final examination. The questions are intended to provoke further thought about issues raised in the chapter. We do not provide solutions, because we anticipate that the best answer to any interesting question will change over the timespan of the reader's career.

Our Focus

In writing this book, we have made a series of conscious decisions that have a strong impact on both its style and its content. At a high level, our focus is to prune, to relate, and to engineer.

Prune Selection of material is an important issue in the design of a compiler construction course today. The sheer volume of information available has grown dramatically over the past decade or two. David Gries' classic book (*Compiler Construction for Digital Computers, John Wiley, 1971*) covers code optimization in a single chapter of less than forty pages. In contrast, Steve Muchnick's recent book (*Advanced Compiler Design and Implementation, Morgan Kauffman, 1997*) devotes thirteen chapters and over five hundred forty pages to the subject, while Bob Morgan's recent book (*Building an Optimizing Compiler, Digital Press, 1998*) covers the material in thirteen chapters that occupy about four hundred pages.

In laying out *Engineering a Compiler*, we have selectively pruned the material to exclude material that is redundant, that adds little to the student's insight and experience, or that has become less important due to changes in languages, in compilation techniques, or in systems architecture. For example, we have omitted operator precedence parsing, the LL(1) table construction algorithm, various code generation algorithms suitable for the PDP-11, and the UNION-FIND-based algorithm for processing Fortran `Equivalence` statements. In their place, we have added coverage of topics that include instruction scheduling, global register allocation, implementation of object-oriented languages, string manipulation, and garbage collection.

Relate Compiler construction is a complex, multifaceted discipline. The solutions chosen for one problem affect other parts of the compiler because they shape the input to subsequent phases and the information available in those phases. Current textbooks fail to clearly convey these relationships.

To make students aware of these relationships, we expose some of them directly and explicitly in the context of practical problems that arise in commonly-used languages. We present several alternative solutions to most of the problems that we address, and we discuss the differences between the solutions and their overall impact on compilation. We try to select examples that are small enough to be grasped easily, but large enough to expose the student to the full complexity of each problem. We reuse some of these examples in several chapters to provide continuity and to highlight the fact that several different approaches can be used to solve them.

Finally, to tie the package together, we provide a couple of questions at the end of each chapter. Rather than providing homework-style questions that have algorithmic answers, we ask exam-style questions that try to engage the student in a process of comparing possible approaches, understanding the tradeoffs between them, and using material from several chapters to address the issue at hand. The questions are intended as a tool to make the reader think, rather than acting as a set of possible exercises for a weekly homework assignment. (We believe that, in practice, few compiler construction courses assign weekly homework. Instead, these courses tend to assign laboratory exercises that provide the student with hands-on experience in language implementation.)

Engineer Legendary compilers, such as the Bliss-11 compiler or the Fortran-H compiler, have done several things well, rather than doing everything in moderation. We want to show the design issues that arise at each stage and how different solutions affect the resulting compiler and the code that it generates.

For example, a generation of students studied compilation from books that assume stack allocation of activation records. Several popular languages include features that make stack allocation less attractive; a modern textbook should present the tradeoffs between keeping activation records on the stack, keeping them in the heap, and statically allocating them (when possible).

When the most widely used compiler-construction books were written, most computers supported byte-oriented load and store operations. Several of them had hardware support for moving strings of characters from one memory location to another (the move character long instruction – `mvcl`). This simplified the treatment of character strings, allowing them to be treated as vectors of bytes (sometimes, with an implicit loop around the operation). Thus, compiler books scarcely mentioned support for strings.

Some RISC machines have weakened support for sub-word quantities; the compiler must worry about alignment; it may need to mask a character into a word using boolean operations. The advent of register-to-register load-store machines eliminated instructions like `mvcl`; today's RISC machine expects the compiler to optimize such operations and work together with the operating system to perform them efficiently.

Trademark Notices

In the text, we have used the registered trademarks of several companies.

IBM is a trademark of International Business Machines, Incorporated.

Intel and IA-64 are trademarks of Intel Corporation.

370 is a trademark of International Business Machines, Incorporated.

MC68000 is a trademark of Motorola, Incorporated.

PostScript is a registered trademark of Adobe Systems.

PowerPC is a trademark of (?Motorola or IBM?)

PDP-11 is a registered trademark of Digital Equipment Corporation, now a part of Compaq Computer.

Unix is a registered trademark of someone or other (maybe Novell).

VAX is a registered trademark of Digital Equipment Corporation, now a part of Compaq Computer.

Java may or may not be a registered trademark of SUN Microsystems, Incorporated.

Acknowledgements

We particularly thank the following people who provided us with direct and useful feedback on the form, content, and exposition of this book: Preston Briggs, Timothy Harvey, L. Taylor Simpson, Dan Wallach.

Contents

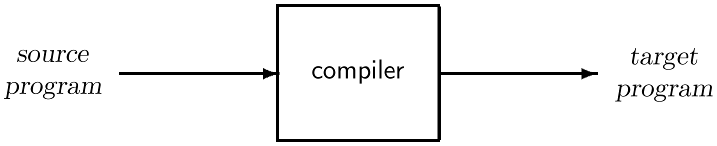
Chapter 1

An Overview of Compilation

1.1 Introduction

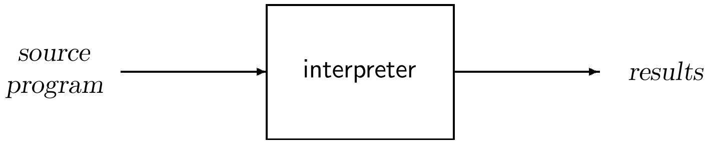
The role of computers in daily life is growing each year. Modern microprocessors are found in cars, microwave ovens, dishwashers, mobile telephones, GPSS navigation systems, video games and personal computers. Each of these devices must be programmed to perform its job. Those programs are written in some "programming" language – a formal language with mathematical properties and well-defined meanings – rather than a natural language with evolved properties and many ambiguities. Programming languages are designed for expressiveness, conciseness, and clarity. A program written in a programming language must be translated before it can execute directly on a computer; this translation is accomplished by a software system called a *compiler*. This book describes the mechanisms used to perform this translation and the issues that arise in the design and construction of such a translator.

A compiler is just a computer program that takes as input an executable program and produces as output an equivalent executable program.



In a traditional compiler, the input language is a programming language and the output language is either assembly code or machine code for some computer system. However, 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. Since 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 results of executing the specification.



Interpreters and compilers have much in common. From an implementation perspective, interpreters and compilers perform many of the same tasks. For example, both must analyze the source code for errors in either syntax or meaning. However, interpreting the code to produce a result is quite different from emitting a translated program that can be executed to produce the results. This book focuses on the problems that arise in building compilers. However, an implementor of interpreters may find much of the material relevant.

The remainder of this chapter presents a high-level overview of the translation process. It addresses both the problems of translation—what issues must be decided along the way—and the structure of a modern compiler—where in the process each decision should occur. Section 1.2 lays out two fundamental principles that every compiler must follow, as well as several other properties that might be desirable in a compiler. Section 1.3 examines the tasks that are involved in translating code from a programming language to code for a target machine. Section 1.4 describes how compilers are typically organized to carry out the tasks of translation.

1.2 Principles and Desires

Compilers are engineered objects—software systems built with distinct goals in mind. In building a compiler, the compiler writer makes myriad design decisions. Each decision has an impact on the resulting compiler. While many issues in compiler design are amenable to several different solutions, there are two principles that should not be compromised. The first principle that a well-designed compiler must observe is inviolable.

The compiler must preserve the meaning of the program being compiled

> The code produced by the compiler must faithfully implement the "meaning" of the source-code program being compiled. If the compiler can take liberties with meaning, then it can always generate the same code, independent of input. For example, the compiler could simply emit a `nop` or a `return` instruction.

The second principle that a well-designed compiler must observe is quite practical.

The compiler must improve the source code in some discernible way.

> If the compiler does not improve the code in some way, why should anyone invoke it? A traditional compiler improves the code 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. Some compilers produce output programs in the same language as their input; we call these "source-to-source" translators. In general, these systems try to restate the program in a way that will lead, eventually, to an improvement.

These are the two fundamental principles of compiler design.

This is an exciting era in the design and implementation of compilers. In the 1980's almost all compilers were large, monolithic systems. They took as input one of a handful of languages—typically Fortran or C—and produced assembly code for some particular computer. The assembly code was pasted together with the code produced by other compiles—including system libraries and application libraries—to form an executable. The executable was stored on a disk; at the appropriate time, the final code was moved from disk to main memory and executed.

Today, compiler technology is being applied in many different settings. These diverse compilation and execution environments are challenging the traditional image of a monolithic compiler and forcing implementors to reconsider many of the design tradeoffs that seemed already settled.

- Java has reignited interest in techniques like "just-in-time" compilation and "throw-away code generation." Java applets are transmitted across the Internet in some internal form, called Java bytecodes; the bytecodes are then interpreted or compiled, loaded, and executed on the target machine. The performance of the tool that uses the applet depends on the total time required to go from bytecodes on a remote disk to a completed execution on the local machine.

- Many techniques developed for large, monolithic compilers are being applied to analyze and improve code at link-time. In these systems, the compiler takes advantage of the fact that the entire program is available at link-time. The "link-time optimizer" analyzes the assembly code to derive knowledge about the run-time behavior of the program and uses that knowledge to produce code that runs faster.

- Some compilation techniques are being delayed even further—to run-time. Several recent systems invoke compilers during program execution to generate customized code that capitalizes on facts that cannot be known any earlier. If the compile time can be kept small and the benefits are large, this strategy can produce noticeable improvements.

In each of these settings, the constraints on time and space differ, as do the expectations with regard to code quality.

The priorities and constraints of a specific project may dictate specific solutions to many design decisions or radically narrow the set of feasible choices. Some of the issues that may arise are:

1. *Speed:* At any point in time, there seem to be applications that need more performance than they can easily obtain. For example, our ability to simulate the behavior of digital circuits, like microprocessors, always lags far behind the demand for such simulation. Similarly, large physical problems such as climate modeling have an insatiable demand for computation. For these applications, the runtime performance of the compiled code is a critical issue. Achieving predictably good performance requires additional analysis and transformation at compile-time, typically resulting in longer compile times.

2. *Space:* Many applications impose tight restrictions on the size of compiled code. Usually, the constraints arise from either physical or economic factors; for example, power consumption can be a critical issue for any battery-powered device. Embedded systems outnumber general purpose computers; many of these execute code that has been committed permanently to a small "read-only memory" (ROM). Executables that must be transmitted between computers also place a premium on the size of compiled code. This includes many Internet applications, where the link between computers is slow relative to the speed of computers on either end.

3. *Feedback:* When the compiler encounters an incorrect program, it must report that fact back to the user. The amount of information provided to the user can vary widely. For example, the early Unix compilers often produced a simple and uniform message "`syntax error`." At the other end of the spectrum the Cornell PL/C system, which was designed as a "student" compiler, made a concerted effort to correct every incorrect program and execute it [23].

4. *Debugging:* Some transformations that the compiler might use to speed up compiled code can obscure the relationship between the source code and the target code. If the debugger tries to relate the state of the broken executable back to the source code, the complexities introduced by radical program transformations can cause the debugger to mislead the programmer. Thus, both the compiler writer and the user may be forced to choose between efficiency in the compiled code and transparency in the debugger. This is why so many compilers have a "debug" flag that causes the compiler to generate somewhat slower code that interacts more cleanly with the debugger.

5. *Compile-time efficiency:* Compilers are invoked frequently. Since the user usually waits for the results, compilation speed can be an important issue. In practice, no one likes to wait for the compiler to finish. Some users will be more tolerant of slow compiles, especially when code quality is a serious issue. However, given the choice between a slow compiler and a fast compiler that produces the same results, the user will undoubtedly choose the faster one.

Before reading the rest of this book, you should write down a prioritized list of the qualities that you want in a compiler. You might apply the ancient standard from software engineering—evaluate features as if you were paying for them with your own money! Examining your list will tell you a great deal about how you would make the various tradeoffs in building your own compiler.

1.3 High-level View of Translation

To gain a better understanding of the tasks that arise in compilation, consider what must be done to generate executable code for the following expression:

$$\mathtt{w} \leftarrow \mathtt{w} \times \mathtt{2} \times \mathtt{x} \times \mathtt{y} \times \mathtt{z}.$$

Let's follow the expression through compilation to discover what facts must be discovered and what questions must be answered.

1.3.1 Understanding the Input Program

The first step in compiling our expression is to determine whether or not

$$\mathtt{w} \leftarrow \mathtt{w} \times \mathtt{2} \times \mathtt{x} \times \mathtt{y} \times \mathtt{z}.$$

is a legal sentence in the programming language. While it might be amusing to feed random words to an English to Italian translation system, the results are unlikely to have meaning. A compiler must determine whether or not its input constitutes a well-constructed sentence in the source language. If the input is well-formed, the compiler can continue with translation, optimization, and code generation. If it is not, the compiler should report back to the user with a clear error message that isolates, to the extent possible, the problem with the sentence.

Syntax In a compiler, this task is called *syntax analysis.* To perform syntax analysis efficiently, the compiler needs:

1. a formal definition of the source language,

2. an efficient membership test for the source language, and

3. a plan for how to handle illegal inputs.

Mathematically, the source language is a set, usually infinite, of strings defined by some finite set of rules. The compiler's front end must determine whether the source program presented for compilation is, in fact, an element in that set of valid strings. In engineering a compiler, we would like to answer this membership question efficiently. If the input program is not in the set, and therefore not in the language, the compiler should provide useful and detailed feedback that explains where the input deviates from the rules.

To keep the set of rules that define a language small, the rules typically refer to words by their syntactic categories, or parts-of-speech, rather than individual words. In describing English, for example, this abstraction allows us to state that many sentences have the form

sentence → *subject verb object period*

rather than trying to enumerate the set of all sentences. For example, we use a syntactic variable, *verb*, to represent all possible verbs. With English, the reader generally recognizes many thousand words and knows the possible parts-of-speech that each can fulfill. For an unfamiliar string, the reader consults a dictionary. Thus, the syntactic structure of the language is based on a set of rules, or a grammar, and a system for grouping characters together to form words and for classifying those words into their syntactic categories.

This description-based approach to specifying a language is critical to compilation. We cannot build a software system that contains an infinite set of rules, or an infinite set of sentences. Instead, we need a finite set of rules that can generate (or specify) the sentences in our language. As we will see in the next two chapters, the finite nature of the specification does not limit the expressiveness of the language.

To understand whether the sentence "Compilers are engineered objects." is, in fact, a valid English sentence, we first establish that each word is valid. Next, each word is replaced by its syntactic category to create a somewhat more abstract representation of the sentence–

noun verb adjective noun period

Finally, we try to fit this sequence of abstracted words into the rules for an English sentence. A working knowledge of English grammar might include the following rules:

1	*sentence*	→	*subject verb object*
2	*subject*	→	*noun*
3	*subject*	→	*modifier noun*
4	*object*	→	*noun*
5	*object*	→	*modifier noun*
6	*modifier*	→	*adjective*
7	*modifier*	→	*adjectival phrase*
	...		

Here, the symbol → reads "derives" and means that an instance of the right hand side can be abstracted to the left hand side. By inspection, we can discover the following *derivation* for our example sentence.

Rule	*Prototype Sentence*
—	*sentence*
1	*subject verb object period*
2	*noun verb object period*
5	*noun verb modifier noun period*
6	*noun verb adjective noun period*

At this point, the prototype sentence generated by the derivation matches the abstract representation of our input sentence. Because they match, at this level of abstraction, we can conclude that the input sentence is a member of the language described by the grammar. This process of discovering a valid derivation for some stream of tokens is called parsing.

If the input is not a valid sentence, the compiler must report the error back to the user. Some compilers have gone beyond diagnosing errors; they have attempted to correct errors. When an error-correcting compiler encounters an invalid program, it tries to discover a "nearby" program that is well-formed. The classic game to play with an error-correcting compiler is to feed it a program written in some language it does not understand. If the compiler is thorough, it will faithfully convert the input into a syntactically correct program and produce executable code for it. Of course, the results of such an automatic (and unintended) transliteration are almost certainly meaningless.

Meaning A critical observation is that syntactic correctness depended entirely on the parts of speech, not the words themselves. The grammatical rules are oblivious to the difference between the noun "compiler" and the noun "tomatoes". Thus, the sentence "Tomatoes are engineered objects." is grammatically indistinguishable from "Compilers are engineered objects.", even though they have significantly different meanings. To understand the difference between these two sentences requires contextual knowledge about both compilers and vegetables.

Before translation can proceed, the compiler must determine that the program has a well-defined meaning. Syntax analysis can determine that the sentences are well-formed, at the level of checking parts of speech against grammatical rules. Correctness and meaning, however, go deeper than that. For example, the compiler must ensure that names are used in a fashion consistent with their declarations; this requires looking at the words themselves, not just at their syntactic categories. This analysis of meaning is often called either *semantic analysis* or *context-sensitive analysis*. We prefer the latter term, because it emphasizes the notion that the correctness of some part of the input, at the level of meaning, depends on the context that both precedes it and follows it.

A well-formed computer program specifies some computation that is to be performed when the program executes. There are many ways in which the expression

$$\texttt{w} \leftarrow \texttt{w} \times 2 \times \texttt{x} \times \texttt{y} \times \texttt{z}$$

might be ill-formed, beyond the obvious, syntactic ones. For example, one or more of the names might not be defined. The variable x might not have a value when the expression executes. The variables y and z might be of different types that cannot be multiplied together. Before the compiler can translate the expression, it must also ensure that the program has a well-defined meaning, in the sense that it follows some set of additional, extra-grammatical rules.

Compiler Organization The compiler's *front end* performs the analysis to check for syntax and meaning. For the restricted grammars used in programming languages, the process of constructing a valid derivation is easily automated. For efficiency's sake, the compiler usually divides this task into *lexical analysis,* or *scanning,* and *syntax analysis,* or *parsing.* The equivalent skill for "natural" languages is sometimes taught in elementary school. Many English grammar books teach a technique called "diagramming" a sentence—drawing a pictorial representation of the sentence's grammatical structure. The compiler accomplishes this by applying results from the study of formal languages [1]; the problems are tractable because the grammatical structure of programming languages is usually more regular and more constrained than that of a natural language like English or Japanese.

Inferring meaning is more difficult. For example, are w, x, y, and z declared as variables and have they all been assigned values previously? Answering these questions requires deeper knowledge of both the surrounding context and the source language's definition. A compiler needs an efficient mechanism for determining if its inputs have a legal meaning. The techniques that have been used to accomplish this task range from high-level, rule-based systems through *ad hoc* code that checks specific conditions.

Chapters 2 through 5 describe the algorithms and techniques that a compiler's front end uses to analyze the input program and determine whether it is well-formed, and to construct a representation of the code in some internal form. Chapter 6 and Appendix B, explore the issues that arise in designing and implementing the internal structures used throughout the compiler. The front end builds many of these structures.

1.3.2 Creating and Maintaining the Runtime Environment

Our continuing example concisely illustrates how programming languages provides their users with abstractions that simplify programming. The language defines a set of facilities for expressing computations; the programmer writes code that fits a model of computation implicit in the language definition. (Implementations of QuickSort in scheme, Java, and Fortran would, undoubtedly, look quite different.) These abstractions insulate the programmer from low-level details of the computer systems they use. One key role of a compiler is to put in place mechanisms that efficiently create and maintain these illusions. For example, assembly code is a convenient fiction that allows human beings to read and write short mnemonic strings rather than numerical codes for operations;

somehow this is more intuitive to most assembly programmers. This particular illusion—that the computer understands alphabetic names rather than binary numbers—is easily maintained by a lookup-table in a symbolic assembler.

The example expression showcases one particular abstraction that the compiler maintains, symbolic names. The example refers to values with the names w, x, y, and z. These names are not just values; a given name can take on multiple values as the program executes. For example, w is used on both the right-hand side and the left-hand side of the assignment. Clearly, w has one value before execution of the expression and another afterwards (unless $\texttt{x} \times \texttt{y} \times \texttt{z} \cong \frac{1}{2}$). Thus, w refers to whatever value is stored in some named location, rather than a specific value, such as 15.

The memories of modern computers are organized by numerical addresses, not textual names. Within the address space of an executing program, these addresses correspond uniquely to storage locations. In a source-level program, however, the programmer may create many distinct variables that use the same name. For example, many programs define the variables i, j, and k in several different procedures; they are common names for loop index variables. The compiler has responsibility for mapping each use of the name j to the appropriate instance of j and, from there, into the storage location set aside for that instance of j. Computers do not have this kind of name space for storage locations; it is an abstraction created by the language designer and maintained by the compiler-generated code and its run-time environment.

To translate $\texttt{w} \leftarrow \texttt{w} \times \texttt{2} \times \texttt{x} \times \texttt{y} \times \texttt{z}$, the compiler must assign some storage location to each name. (We will assume, for the moment, that the constant two needs no memory location since it is a small integer and can probably be obtained using a *load immediate* instruction.) This might be done in memory, as in

0	w	x	y	z

or, the compiler might elect to keep the named variables in machine registers with a series of assignments:

$$r_1 \leftarrow \texttt{w};\ r_2 \leftarrow \texttt{x};\ r_3 \leftarrow \texttt{y};\ \text{and}\ r_4 \leftarrow \texttt{z};$$

The compiler must choose, based on knowledge of the surrounding context, a location for each named value. Keeping w in a register will likely lead to faster execution; if some other statement assigns w's address to a pointer, the compiler would need to assign w to an actual storage location.

Names are just one abstraction maintained by the compiler. To handle a complete programming language, the compiler must create and support a variety of abstractions, Procedures, parameters, names, lexical scopes, and control-flow operations are all abstractions designed into the source language that the compiler creates and maintains on the target machine (with some help from the other system software). Part of this task involves the compiler emitting the appropriate instructions at compile time; the remainder involves interactions between that compiled code and the run-time environment that supports it.

```
loadAI   r_arp, @w         ⇒ r_w        // load 'w'
loadI    2                 ⇒ r_2        // constant 2 into r_2
loadAI   r_arp, load 'x'
loadAI   r_arp, load 'y'
loadAI   r_arp, load 'z'
mult     r_w, r_2          ⇒ r_w        // r_w ← w×2
mult     r_w, r_x          ⇒ r_w        // r_w ← (w×2) × x
mult     r_w, r_y          ⇒ r_w        // r_w ← (w×2×x) × y
mult     r_w, r_z          ⇒ r_w        // r_w ← (w×2×x×y) × z
storeAI  r_w               ⇒ r_arp, 0   // write r_w back to 'w'
```

Figure 1.1: Example in ILOC

Thus, designing and implementing a compiler involves not only translation from some source language to a target language, but also the construction of a set of mechanisms that will create and maintain the necessary abstractions at run-time. These mechanisms must deal with the layout and allocation of memory, with the orderly transfer of control between procedures, with the transmission of values and the mapping of name spaces at procedure borders, and with interfaces to the world outside the compiler's control, including input and output devices, the operating system, and other running programs.

Chapter 7 explores the abstractions that the compiler must maintain to bridge the gap between the programming model embodied in the source language and the facilities provided by the operating system and the actual hardware. It describes algorithms and techniques that the compilers use to implement the various fictions contained in the language definitions. It explores some of the issues that arise on the boundary between the compiler's realm and that of the operating system.

1.3.3 Creating the Output Program

So far, all of the issues that we have addressed also arise in interpreters. The difference between a compiler and an interpreter is that the compiler emits executable code as its output, while the interpreter produces the result of executing that code. During code generation, the compiler traverses the internal data structures that represent the code and it emits equivalent code for the target machine. It must select instructions to implement each operation that appears in the code being compiled, decide when and where to move values between registers and memory, and choose an execution order for the instructions that both preserves meaning and avoids unnecessary hardware stalls or interlocks. (In contrast, an interpreter would traverse the internal data structures and simulate the execution of the code.)

Instruction Selection As part of code generation, the compiler must select a sequence of machine instructions to implement the operations expressed in the code being compiled. The compiler might choose the instructions shown in

Digression: About ILOC

Throughout the book, low-level examples are written in an notation that we call ILOC—an acronym that William LeFebvre derived from "intermediate language for an optimizing compiler." Over the years, this notation has undergone many changes. The version used in this book is described in detail in Appendix A.

Think of ILOC as the assembly language for a simple RISC machine. It has a standard complement of operations Most operations take arguments that are registers. The memory operations, `load`s and `store`s, transfer values between memory and the registers. To simplify the exposition in the text, most examples assume that all data is integer data.

Each operation has a set of operands and a target. The operation is written in five parts: an operation name, a list of operands, a separator, a list of targets, and an optional comment. Thus, to add registers 1 and 2, leaving the result in register 3, the programmer would write

$$\texttt{add}\quad \texttt{r}_1\texttt{,r}_2 \;\Rightarrow\; \texttt{r}_3 \quad \texttt{// example instruction}$$

The separator, ⇒, precedes the target list. It is a visual reminder that information flows from left to right. In particular, it disambiguates cases like `load` and `store`, where a person reading the assembly-level text can easily confuse operands and targets.

Figure 1.1 to implement

$$\texttt{w} \leftarrow \texttt{w} \times 2 \times \texttt{x} \times \texttt{y} \times \texttt{z}$$

on the ILOC virtual machine. Here, we have assumed the memory layout shown earlier, where `w` appears at memory address zero.

This sequence is straight forward. It loads all of the relevant values into registers, performs the multiplications in order, and stores the result back to the memory location for `w`. Notice that the registers have unusual names, like $\texttt{r}_w$ to hold `w` and $\texttt{r}_{arp}$ to hold the address where the data storage for our named values begins. Even with this simple sequence, however, the compiler makes choices that affect the performance of the resulting code. For example, if an *immediate multiply* is available, the instruction $\texttt{mult r}_w\texttt{, r}_2 \Rightarrow \texttt{r}_w$ could be replaced with $\texttt{multI r}_w\texttt{, 2} \Rightarrow \texttt{r}_w$, eliminating the need for the instruction $\texttt{loadI 2} \Rightarrow \texttt{r}_2$ and decreasing the number of registers needed. If multiplication is slower than addition, the instruction could be replaced with $\texttt{add r}_w\texttt{, r}_w \Rightarrow \texttt{r}_w$, avoiding the `loadI` and its use of $\texttt{r}_2$ as well as replacing the `mult` with a faster `add` instruction.

Register Allocation In picking instructions, we ignored the fact that the target machine has a finite set of registers. Instead, we assumed that "enough" registers existed. In practice, those registers may or may not be available; it depends on how the compiler has treated the surrounding context.

In register allocation, the compiler decides which values should reside in the registers of the target machine, at each point in the code. It then modifies the code to reflect its decisions. If, for example, the compiler tried to minimize the number of registers used in evaluating our example expression, it might generate the following code

```
loadAI   r_arp, @w  ⇒ r_1          // load 'w'
add      r_1, r_1   ⇒ r_1          // r_1 ← w × 2
loadAI   r_arp, @x  ⇒ r_2          // load 'x'
mult     r_1, r_2   ⇒ r_1          // r_1 ← (w×2) × x
loadAI   r_arp, @y  ⇒ r_2          // load 'y'
mult     r_1, r_2   ⇒ r_1          // r_1 ← (w×2×x) × y
loadAI   r_arp, @z  ⇒ r_2          // load 'z'
mult     r_1, r_2   ⇒ r_1          // r_1 ← (w×2×x×y) × z
storeAI  r_1        ⇒ r_arp, @w    // write r_w back to 'w'
```

This sequence uses two registers, plus r_{arp}, instead of five.

Minimizing register use may be counter productive. If, for example, any of the named values, w, x, y, or z, 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 w × 2, it might be better to preserve that value in a register than to recompute it later. This would increase demand for registers, but eliminate a later instruction.

In general, the problem of allocating values to registers is NP-complete. Thus, we should not expect the compiler to discover optimal solutions to the problem, unless we allow exponential time for some compilations. In practice, compilers use approximation techniques to discover good solutions to this problem; the solutions may not be optimal, but the approximation techniques ensure that some solution is found in a reasonable amount of time.

Instruction Scheduling In generating code for a target machine, the compiler should be aware of that machine's specific performance constraints. For example, we mentioned that an addition might be faster than a multiplication; in general, the execution time of the different instructions can vary widely. Memory access instructions (loads and stores) can take many cycles, while some arithmetic instructions, particularly mult, take several. The impact of these longer latency instructions on the performance of compiled code is dramatic.

Assume, for the moment, that a load or store instruction requires three cycles, a mult requires two cycles, and all other instructions require one cycle. With these latencies, the code fragment that minimized register use does not look so attractive. The *Start* column shows the cycle in which the instruction begins execution and the *End* column shows the cycle in which it completes.

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

This nine instruction sequence takes twenty-two cycles to execute.

Many modern processors have the property that they can initiate new instructions while a long-latency instruction executes. As long as the results of a long-latency instruction are not referenced until the instruction completes, execution proceeds normally. If, however, some intervening instruction tries to read the result of the long-latency instruction prematurely, the processor "stalls", or waits until the long-latency instruction completes. Registers are read in the cycle when the instruction starts and written when it ends.

In instruction scheduling, the compiler reorders the instructions in an attempt to minimize the number cycles wasted in stalls. Of course, the scheduler must ensure that the new sequence produces the same result as the original. In many cases, the scheduler can drastically improve on the performance of "naive" code. For our example, a good scheduler might produce

Start	*End*				
1	3	loadAI	r_{arp}, @w	⇒ r_1	// load 'w'
2	4	loadAI	r_{arp}, @x	⇒ r_2	// load 'x'
3	5	loadAI	r_{arp}, @y	⇒ r_3	// load 'y'
4	4	add	r_1, r_1	⇒ r_1	// $r_1 \leftarrow$ w × 2
5	6	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow$ (w×2) × x
6	8	loadAI	r_{arp}, @z	⇒ r_2	// load 'z'
7	8	mult	r_1, r_3	⇒ r_1	// $r_1 \leftarrow$ (w×2×x) × y
9	10	mult	r_1, r_2	⇒ r_1	// $r_1 \leftarrow$ (w×2×x×y) × z
11	13	storeAI	r_1	⇒ r_{arp}, @w	// write r_w back to 'w'

This reduced the time required for the computation from twenty-two cycles to thirteen. It required one more register than the minimal number, but cut the execution time nearly in half. It starts an instruction in every cycle except eight and ten. This schedule is not unique; several equivalent schedules are possible, as are equal length schedules that use one more register.

Instruction scheduling is, like register allocation, a hard problem. In its general form, it is NP-Complete. Because variants of this problem arise in so many fields, it has received a great deal of attention in the literature.

Interactions Most of the truly hard problems that occur in compilation arise during code generation. To make matters more complex, these problems inter-

Digression: Terminology

A careful reader will notice that we use the word "code" in many places where either "program" or "procedure" might naturally fit. This is a deliberate affectation; compilers can be invoked to translate fragments of code that range from a single reference through an entire system of programs. Rather than specify some scope of compilation, we will continue to use the ambiguous term "code."

act. For example, instruction scheduling moves `load` instructions away from the arithmetic operations that depend on them. This can increase the period over which the value is needed and, correspondingly, increase the number of registers needed during that period. Similarly, the assignment of particular values to specific registers can constrain instruction scheduling by creating a "false" dependence between two instructions. (The second instruction cannot be scheduled until the first completes, even though the values in the overlapping register are independent. Renaming the values can eliminate this false dependence, at the cost of using more registers.)

Chapters 8 through 11 describe the issues that arise in code generation and present a variety of techniques to address them. Chapter 8 creates a base of knowledge for the subsequent work. It discusses "code shape," the set of choices that the compiler makes about how to implement a particular source language construct. Chapter 9 builds on this base by discussing algorithms for instruction selection—how to map a particular code shape into the target machine's instruction set. Chapter 10 looks at the problem of deciding which values to keep in registers and explores algorithms that compilers use to make these decisions. Finally, because the order of execution can have a strong impact on the performance of compiled code, Chapter 11 delves into algorithms for scheduling instructions.

1.3.4 Improving the Code

Often, a compiler can use contextual knowledge to improve the quality of code that it generates for a statement. If, as shown on the left side of Figure 1.2, the statement in our continuing example was embedded in a loop, the contextual information might let the compiler significantly improve the code. The compiler could recognize that the subexpression `2` $\times$ `x` $\times$ `y` is invariant in the loop – that is, its value does not change between iterations. Knowing this, the compiler could rewrite the code as shown on the right side of the figure. The transformed code performs many fewer operations in the loop body; if the loop executes more than once, this should produce faster code.

This process of analyzing code to discover facts from context and using that knowledge to improve the code is often called *code optimization.* Roughly speaking, optimization consists of two distinct activities: analyzing the code to understand its runtime behavior and transforming the code to capitalize on knowledge derived during analysis. These techniques play a critical role in the

```
$x \leftarrow \cdots$
$y \leftarrow \cdots$
$w \leftarrow 1$

for $i = 1$ to $n$
    read $z$
    $w \leftarrow w \times 2 \times x \times y \times z$
    end
```

Surrounding context

```
$x \leftarrow \cdots$
$y \leftarrow \cdots$
$t_i \leftarrow 2 \times x \times y$
$w \leftarrow 1$
for $i = 1$ to $n$
    read $z$
    $w \leftarrow w \times z \times t_i$
    end
```

Improved code

Figure 1.2: Context makes a difference

performance of compiled code; the presence of a good optimizer can change the kind of code that the rest of the compiler should generate.

Analysis Compilers use several kinds of analysis to support transformations. *Data-flow analysis* involves reasoning, at compile-time, about the flow of values at runtime. Data-flow analyzers typically solve a system of simultaneous set equations that are derived from the structure of the code being translated. *Dependence analysis* uses number-theoretic tests to reason about the values that can be assumed by subscript expressions. It is used to disambiguate references to elements of arrays and indexed structures.

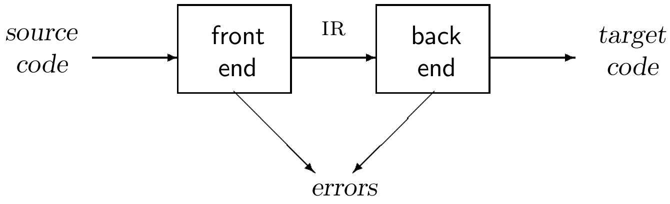
Transformation Many distinct transformations have been invented that try to improve the time or space requirements of executable code. Some of these, like discovering loop-invariant computations and moving them to less frequently executed locations, improve the running time of the program. Others make the code itself 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 a completely separate book.

The final part of the book introduces the techniques that compilers use to analyze and improve programs. Chapter 13 describes some of the methods that compilers use to predict the runtime behavior of the code being translated. Chapter 14 then presents a sampling of the transformations that compilers apply to capitalize on knowledge derived from analysis.

1.4 Compiler Structure

Understanding the issues involved in translation is different than knowing their solutions. The community has been building compilers since 1955; over those years, we have learned a number of lessons about how to structure a compiler. At the start of this chapter, the compiler was depicted as a single box that translated a source program into a target program. Reality is, of course, more complex than that simple pictogram.

The discussion in Section 1.3 suggested a dichotomy between the task of understanding the input program and the task of mapping its functionality onto the target machine. Following this line of thought leads to a compiler that is decomposed into two major pieces, a *front end* and a *back end*.



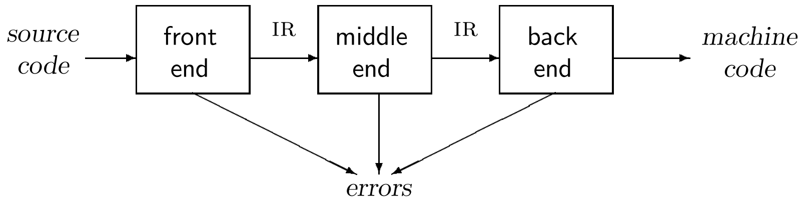
The decision to let the structure reflect the separate nature of the two tasks has several major implications for compiler design.

First, the compiler must have some structure that encodes its knowledge of the code being compiled; this *intermediate representation* (IR) or *intermediate language* becomes the definitive representation of the code for the back end. Now, the task of the front end is to ensure that the source program is well formed and to map that code into the IR, and the task of the back end is to map the IR onto the target machine. Since the back end only processes IR created by the front end, it can assume that the IR contains no errors.

Second, the compiler now makes multiple passes over the code before committing itself to target code. This should lead to better code; the compiler can, in effect, study the code in its first pass and record relevant details. Then, in the second pass, it can use these recorded facts to improve the quality of translation. (This idea is not new. The original FORTRAN compiler made several passes over the code [3]. In a classic 1961 paper, Floyd proposed that the compiler could generate better code for expressions if it made two passes over the code [31].) To achieve this, however, the knowledge derived in the first pass must be recorded in the IR, where the second pass can find it.

Finally, the two pass structure may simplify the process of retargeting the compiler. We can easily envision constructing multiple back ends for a single front end; doing so would produce compilers that accepted the same language but targeted different machines. This assumes that the same IR program is appropriate for both target machines; in practice, some machine-specific details usually find their way into the IR.

The introduction of an IR into the compiler makes possible further passes over the code. These can be implemented as transformers that take as input an IR program and produce an equivalent, but improved, IR program. (Notice that these transformers are, themselves, compilers according to our definition in Section 1.1.) These transformers are sometimes called *optimizations*; they can be grouped together to form an *optimizer* or a *middle end*. This produces a structure that looks like:



We will call this a *three-pass compiler*; it is often called an *optimizing compiler*. Both are misnomers. Almost all compilers have more than three passes. Still, the conceptual division into front end, middle end, and back end is useful. These three parts of the compiler have quite different concerns. Similarly, the term "optimization" implies that the compiler discovers an optimal solution to some problem. Many of the problems in that arise in trying to improve compiled code are so complex that they cannot be solved to optimality in a reasonable amount of time. Furthermore, the actual speed of the compiled code depends on interactions among all of the techniques applied in the optimizer and the back-end. Thus, when a single technique can be proved optimal, its interactions with other techniques can produce less than optimal results. As a result, a good optimizing compiler can improve the quality of the code, relative to an unoptimized version. It will often fail to produce optimal code.

The middle end can be a monolithic structure that applies one or more techniques to improve the code, or it can be structured as a series of individual passes that each read and write IR. The monolithic structure may be more efficient, in that it avoids lots of input and output activity. The multi-pass structure may lend itself to a less complex implementation and a simpler approach to debugging the compiler. The choice between these two approaches depends on the constraints under which the compiler is built and operates.

1.5 Summary and Perspective

A compiler's primary mission is to translate its input program into an equivalent output program. Many different programs qualify as compilers. Most of these can be viewed as either two pass or three pass compilers. They have a front end that deals with the syntax and meaning of the input language and a back end that deals with the constraints of the output language. In between, they may have a section that transforms the program in an attempt to "improve" it.

Different projects, of course, aim for different points in the compiler design space. A compiler that translates C code for embedded applications like automobiles, telephones, and navigation systems, might be concerned about the size of the compiled code, since the code will be burned into some form of read-only memory. On the other hand, a compiler that sits inside the user-interface of a network browser and translates compressed application code to drive the display might be designed to minimize the sum of compile time plus execution time.

Questions

1. In designing a compiler, you will face many tradeoffs. What are the five qualities that you, as a user, consider most important in a compiler that you purchased? Does that list change when you are the compiler writer? What does your list tell you about a compiler that you would implement?

2. 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 cellular telephone

   (c) a compiler used in the introductory programming course at a high school

   (d) a compiler used to build wind-tunnel simulations that run on a massively parallel processors (where all the processors are identical)

   (e) a compiler that targets numerically-intensive programs to a large network of diverse machines

Chapter 2

Lexical Analysis

2.1 Introduction

The scanner takes as input a stream of characters and produces as output a stream of words, along with their associated syntactic categories. It aggregates letters together to form words and applies a set of rules to determine whether or not the word is legal in the source language and, if so, its syntactic category. This task can be done quickly and efficiently using a specialized recognizer.

This chapter describes the mathematical tools and the programming techniques that are commonly used to perform lexical analysis. Most of the work in scanner construction has been automated; indeed, this is a classic example of the application of theoretical results to solve an important and practical problem—specifying and recognizing patterns. The problem has a natural mathematical formulation. The mathematics leads directly to efficient implementation schemes. The compiler writer specifies the lexical structure of the language using a concise notation and the tools transform that specification into an efficient executable program. These techniques have led directly to useful tools in other settings, like the Unix tool `grep` and the regular-expression pattern matching found in many text editors and word-processing tools. Scanning is, essentially, a solved problem.

Scanners look at a stream of characters and recognize words. The rules that govern the lexical structure of a programming language, sometimes called its *micro-syntax*, are simple and regular. This leads to highly efficient, specialized recognizers for scanning. Typically, a compiler's front end has a scanner to handle its micro-syntax and a parser for its context-free syntax, which is more complex to recognize. This setup is shown in Figure 2.1. Separating micro-syntax from syntax simplifies the compiler-writer's life in three ways.

- The description of syntax used in the parser is written in terms of words and syntactic categories, rather than letters, numbers, and blanks. This lets the parser ignore irrelevant issues like absorbing extraneous blanks, newlines, and comments. These are hidden inside the scanner, where they

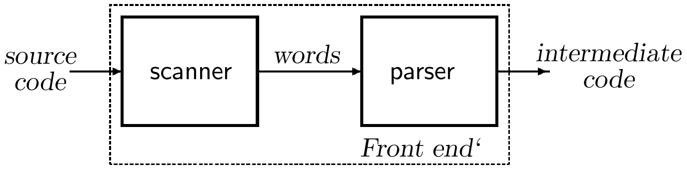


Figure 2.1: Structure of a typical front end

are handled cleanly and efficiently.

- Scanner construction is almost completely automated. The lexical rules are encoded in a formal notation and fed to a scanner generator. The result is an executable program that produces the input for the parser. Scanners generated from high-level specifications are quite efficient.
- Every rule moved into the scanner shrinks the parser. Parsing is harder than scanning; the amount of code in a parser grows as the grammar grows. Since parser construction requires more direct intervention from the programmer, shrinking the parser reduces the compiler-writer's effort.

As a final point, well-implemented scanners have lower overhead (measured by instructions executed per input symbol) than well-implemented parsers. Thus, moving work into the scanner improves the performance of the entire front end.

Our goal for this chapter is to develop the notations for specifying lexical patterns and the techniques to convert those patterns directly into an executable scanner. Figure 2.2 depicts this scenario. This technology should allow the compiler writer to specify the lexical properties at a reasonably high level and leave the detailed work of implementation to the scanner generator—without sacrificing efficiency in the final product, the compiler.

First, we will introduce a notation, called regular expressions, that works well for specifying regular expressions. We will explore the properties of regular expressions and their relationship to a particular kind of recognizer, called a finite automaton. Next, we will develop the techniques and methods that allow us to automate the construction of efficient scanners from regular expressions. We show some additional results that relate regular expressions and automata, and conclude with an example of how complex the task of lexical analysis can be in Fortran, a language designed before we had a good understanding of the mathematics of lexical analysis.

2.2 Specifying Lexical Patterns

Before we can build a scanner, we need a way of specifying the micro-syntax of the source language—of specifying patterns that describe the words in the language. Some parts are quite easy.

- Punctuation marks like colons, semi-colons, commas, parentheses, and square brackets can be denoted by their unique character representations:

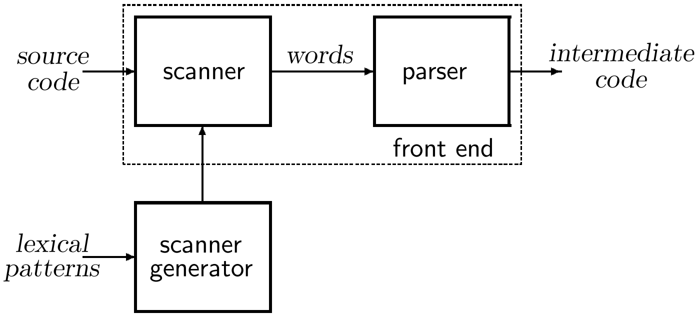


Figure 2.2: Automatic scanner generation

`:   ;   ,   (   )   [   ]`

- Keywords, like `if`, `then`, and `integer` are equally simple. These words have a unique spelling, so we can represent them as literal patterns—we simply write them down.

Some simple concepts have more complicated representations. For example, the concept of a blank might require a small grammar.

$$\begin{array}{lcl} \mathit{WhiteSpace} & \rightarrow & \mathit{WhiteSpace}\ \texttt{blank} \\ & | & \mathit{WhiteSpace}\ \texttt{tab} \\ & | & \texttt{blank} \\ & | & \texttt{tab} \end{array}$$

where `blank` and `tab` have the obvious meanings.

Thus, for many words in the source language, we already have a concise representation—we can simply write down the words themselves. Other parts of a programming language are much harder to specify. For these, we will write down rules that can generate all of the appropriate strings.

Consider, for example, a pattern to specify when a string of characters forms a number. An integer might be described as a string of one or more digits, beginning with a digit other than zero, or as the single digit zero. A decimal number is simply an integer, followed by a decimal point, followed by a string of zero or more digits. (Notice that the part to the left of the decimal point cannot have leading zeros unless it is a zero, while the fractional part must be able to have leading zeros.) Real numbers and complex numbers are even more complicated. Introducing optional signs (+ or -) adds yet more clauses to the rules.

The rules for an identifier are usually somewhat simpler than those for a number. For example, Algol allowed identifier names that consisted of a single alphabetic character, followed by zero or more alphanumeric characters. Many languages include special characters such as the ampersand (`&`), percent sign (`%`), and underscore (`_`) in the alphabet for identifier names.

The complexity of lexical analysis arises from the simple fact that several of the syntactic categories used in a typical programming language contain an effectively infinite set of words. In particular, both numbers and identifiers usually contain sets large enough to make enumeration impractical.[1] To simplify scanner construction, we need to introduce a powerful notation to specify these lexical rules: *regular expressions*.

A regular expression describes a set of strings over the characters contained in some alphabet, Σ, augmented with a character ϵ that represents the empty string. We call the set of strings a *language*. For a given regular expression, r, we denote the language that it specifies as $L(r)$. A regular expression is built up from three simple operations:

Union – the union of two sets R and S, denoted $R \cup S$, is the set $\{s \mid s \in R \text{ or } s \in S\}$.

Concatenation – the concatenation of two sets R and S, denoted RS, is the set $\{st \mid s \in R \text{ and } t \in S\}$. We will sometimes write R^2 for RR, the concatenation of R with itself, and R^3 for RRR (or RR^2).

Closure – the Kleene closure of a set R, denoted R^*, is the set $\bigcup_0^\infty R^i$. This is just the concatenation of L with itself, zero or more times.

It is sometimes convenient to talk about the *positive closure* of R, denoted R^+. It is defined as $\bigcup_1^\infty R^i$, or RR^*.

Using these three operations, we can define the set of regular expressions (REs) over an alphabet Σ.

1. if $a \in \Sigma$, then a is also a RE denoting the set containing only a.
2. if r and s are REs, denoting sets $L(r)$ and $L(s)$ respectively, then

   (r) is a RE denoting $L(r)$

   $r \mid s$ is a RE denoting the union of $L(r)$ and $L(s)$

   rs is a RE denoting the concatenation of $L(r)$ and $L(s)$

   r^* is a RE denoting the Kleene closure of $L(r)$.
3. ϵ is a RE denoting the empty set.

To disambiguate these expressions, we assume that closure has highest precedence, followed by concatenation, followed by union. (Union is sometimes called alternation.)

As a simple example, consider our clumsy English description of the microsyntax of Algol identifiers. As a RE, an identifier might be defined as:

[1] This is not always the case. Dartmouth BASIC, an interpreted language from the early 1960's, only allowed variable names that began with an alphabetic character and had at most one digit following that character. This limited the programmer to two hundred and eighty six variable names. Some implementations simplified the translation by statically mapping each name to one of two hundred and eighty six memory locations.

$$\begin{array}{lll} \mathit{alpha} & \rightarrow & (a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \\ & & \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z) \\ \mathit{digit} & \rightarrow & (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9) \\ \mathit{identifier} & \rightarrow & \mathit{alpha}\ (\mathit{alpha} \mid \mathit{digit})^* \end{array}$$

Here, we introduced names for the subexpressions *alpha* and *digit.* The entire expression could have been written in one-line (with a suitably small font). Where the meaning is clear, we will elide some of the enumerated elements in a definition. Thus, we might write $\mathit{alpha} \rightarrow (a \mid b \mid c \mid \cdots \mid z)$ as a shorthand for the definition of *alpha* given previously. This allows us to write REs more concisely. For example, the Algol-identifier specification now becomes

$$(a \mid b \mid c \mid \cdots \mid z)\ (\ (a \mid b \mid c \mid \cdots \mid z) \mid (0 \mid 1 \mid 2 \mid \cdots \mid 9)\ )^*$$

A similar set of rules can be built up to describe the various kinds of numbers.

$$\begin{array}{lll} \mathit{integer} & \rightarrow & (+ \mid - \mid \epsilon)\ (0 \mid 1 \mid 2 \mid \cdots \mid 9)^+ \\ \mathit{decimal} & \rightarrow & \mathit{integer}\ .\ (0 \mid 1 \mid 2 \mid \cdots \mid 9)^* \\ \mathit{real} & \rightarrow & (\mathit{integer} \mid \mathit{decimal})\ \texttt{E}\ \mathit{integer} \end{array}$$

In the RE *real*, the letter `E` is a delimiter that separates the mantissa from the exponent. (Some programming languages use other letters to denote specific internal formats for floating point numbers.)

Notice that the specification for an integer admits an arbitrary number of leading zeros. We can refine the specification to avoid leading zeros, except for the single, standalone zero required to write the number zero.

$$\mathit{integer} \rightarrow (+ \mid - \mid \epsilon)\ \ (0 \mid\ \ (1 \mid 2 \mid 3 \mid \cdots \mid 9)\ (0 \mid 1 \mid 2 \mid \cdots \mid 9)^*\ )$$

Unfortunately, the rule for *real* relied on leading zeros in its exponent, so we must also rewrite that rule as

$$\begin{array}{lll} \mathit{real} & \rightarrow & (\mathit{integer} \mid \mathit{decimal})\ \texttt{E}\ 0^*\mathit{integer}\text{, or} \\ \mathit{real} & \rightarrow & (\mathit{integer} \mid \mathit{decimal})\ \texttt{E}\ (0 \mid 1 \mid 2 \mid \cdots \mid 9)^+ \end{array}$$

Of course, even more complex examples can be built using the three operations of regular expressions—union, concatenation, and closure.

2.3 Closure Properties of REs

The languages generated by regular expressions have been studied extensively. They have many important properties; some of those properties play an important role in the use of regular expressions to build scanners.

Regular expressions are closed under many operations. For example, given two regular expressions r and $s \in \Sigma^*$, we know that $(r \mid s)$ is a regular expression that represents the language

$$\{w \mid w \in L(r) \text{ or } w \in L(s)\}.$$

This follows from the definition of regular expressions. Because $(r \mid s)$ is, by definition, a regular expression, we say that the set of regular expressions is closed under alternation. From the definition of regular expressions, we know that the set of regular expressions is closed under alternation, under concatenation, and under Kleene closure.

These three properties play a critical role in the use of regular expressions to build scanners. Given a regular expression for each of syntactic categories allowed in the source language, the alternation of those expressions is itself a regular expression that describes the set of all valid words in the language. The fact the regular expressions are closed under alternation assures us that the result is a regular expression. Anything that we can do to the simple regular expression for a single syntactic category will be equally applicable to the regular expression for the entire set of words in the language.

Closure under concatenation allows us to build complex regular expressions from simpler ones by concatenating them together. This property seems both obvious and unimportant. However, it lets us piece together REs in systematic ways. Closure ensures that rs is a RE as long as both r and s are REs. Thus, any techniques that can be applied to either r or s can be applied to rs; this includes constructions that automatically generate recognizer implementations from REs.

The closure property for Kleene closure (or *) allows us to specify particular kinds of infinite sets with a finite pattern. This is critical; infinite patterns are of little use to an implementor. Since the Algol-identifier rule does not limit the length of the name, the rule admits an infinite set of words. Of course, no program can have identifiers of infinite length. However, a programmer might write identifiers of arbitrary, but finite, length. Regular expressions allow us to write concise rules for such a set without specifying a maximum length.

The closure properties for REs introduce a level of abstraction into the construction of micro-syntactic rules. The compiler writer can define basic notions, like *alpha* and *digit*, and use them in other REs. The RE for Algol identifiers

$$\mathit{alpha}\ (\mathit{alpha} \mid \mathit{digit})^*$$

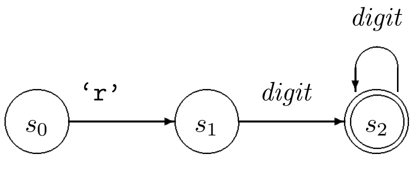
is a RE precisely because of the closure properties. It uses all three operators to combine smaller REs into a more complex specification. The closure properties ensure that the tools for manipulating REs are equally capable of operating on these “composite” REs. Since the tools include scanner generators, this issue plays a direct role in building a compiler.

2.4 Regular Expressions and Finite Automata

Every regular expression, r, corresponds to an abstract machine that recognizes $L(r)$. We call these recognizers *finite automata*. For example, in a lexical analyzer for assembly code, we might find the regular expression

$$\mathtt{r}\ \mathit{digit}\ \mathit{digit}^*$$

The recognizer for this regular expression could be represented, pictorially, as



The diagram incorporates all the information necessary to understand the recognizer or to implement it. Each circle represents a *state*; by convention, the recognizer starts in state s_0. Each arrow represents a *transition*; the label on the transition specifies the set of inputs that cause the recognizer to make that transition. Thus, if the recognizer was in state s_0 when it encountered the input character `r`, it would make the transition to state s_1. In state s_2, any digit takes the recognizer back to s_2. The state s_2 is designated as a final state, drawn with the double circle.

Formally, this recognizer is an example of a finite automaton (FA). An FA is represented, mathematically, as a five-tuple $(Q, \Sigma, \delta, q_0, F)$. Q is the set of states in the automaton, represented by circles in the diagram. Σ is the alphabet of characters that can legally occur in the input stream. Typically, Σ is the union of the edge labels in the diagram. Both Q and Σ must be finite. $\delta : Q \times \Sigma \rightarrow Q$ is the transition function for the FA. It encodes the state changes induced by an input character for each state; δ is represented in the diagram by the labeled edges that connect states. The state $q_0 \in Q$ is the starting state or initial state of the FA. $F \subseteq Q$ is the set of states that are considered final or accepting states. States in F are recognizable in the diagram because they are drawn with a double circle.

Under these definitions, we can see that our drawings of FAs are really pictograms. From the drawing, we can infer Q, Σ, δ, q_0 and F.

The FA *accepts* an input string x if and only if x takes it through a series of transitions that leave it in a final state when x has been completely consumed. For an input string '`r29`', the recognizer begins in s_0. On `r`, it moves to s_1. On `2`, it moves to s_2. On `9`, it moves to s_2. At that point, all the characters have been consumed and the recognizer is in a final state. Thus, it accepts `r29` as a word in $L(\textit{register})$.

More formally, we say that the FA $(Q, \Sigma, \delta, q_0, F)$ accepts the string x, composed of characters $x_1x_2x_3 \ldots x_n$ if and only if

$$\delta(\delta(\ldots\delta(\delta(\delta(q_0, x_1), x_2), x_3) \ldots, x_{n-1}), x_n) \;\in\; F.$$

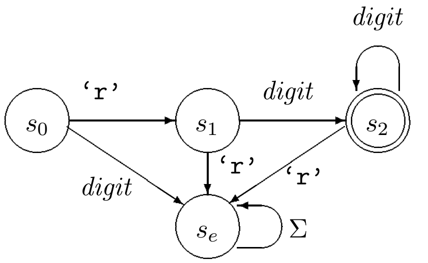
Intuitively, $x_1x_2x_3 \ldots x_n$ corresponds to the labels on a path through the FAs transition diagram, beginning with q_0 and ending in some $q_f \in Q$. At each step, q_i corresponds to the label on the i^{th} edge in the path.

In this more formal model, the FA for register names can be written as

$$
\begin{aligned}
&Q = \{s_0, s_1, s_2\}\\
&\Sigma = \{r, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\\
&\delta = \{(\langle s_0, \mathtt{r}\rangle \rightarrow s_1), (\langle s_1, \mathit{digit}\rangle \rightarrow s_2), (\langle s_2, \mathit{digit}\rangle \rightarrow s_2)\}\\
&q_0 = s_0\\
&F = \{s_2\}
\end{aligned}
$$

Notice how this encodes the diagram. Q contains the states. Σ contains the edge labels. δ encodes the edges.

Error Transitions What happens if we confront the recognizer for register names with the string '`s29`'? State s_0 has no transition for '`s`'. We will assume that any unspecified input leads to an error state s_e. To make the recognizer work, s_e needs a transition back to itself on every character in Σ.

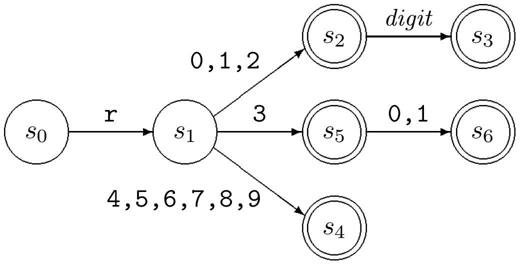


With these additions, if the recognizer reaches s_e, it remains in s_e and consumes all of its input. By convention, we will assume the presence of an error state and the corresponding transitions in every FA, unless explicitly stated. We will not, however, crowd the diagrams by drawing them.

A More Precise Recognizer Our simple recognizer can distinguish between `r29` and `s29`. It does, however, have limits. For example, it will accept `r999`; few computers have been built that have 999 registers! The flaw, however, lies in the regular expression, not in the recognizer. The regular expression specifies the numerical substring as *digit digit*$^+$; this admits arbitrary strings of digits. If the target machine had just thirty-two registers, named `r0` through `r31`, a carefully crafted regular expression might be used, such as:

$$\mathtt{r}\ ((0 \mid 1 \mid 2)\ (\mathit{digit} \mid \epsilon)\ \mid\ (4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)\ \mid\ (3 \mid 30 \mid 31)$$

This expression is more complex than our original expression. It will, however, accept `r0`, `r9`, and `r29`, but not `r999`. The FA for this expression has more states than the FA for the simpler regular expression.



Recall, however, that the FA makes one transition on each input character. Thus, the new FA will make the same number of transitions as the original, even though it checks a more complex micro-syntactic specification. This is a critical property: the cost of operating a FA is proportional to the length of the input string, not to the length or complexity of the regular expression that generated the recognizer. Increased complexity in the regular expression can increase the number of states in the corresponding FA. This, in turn, increases the space needed to represent the FA the cost of automatically constructing it, but the cost of operation is still one transition per input character.

Of course, the RE for register names is fairly complex and counter-intuitive. An alternative RE 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 expression is conceptually simpler, but much longer than the previous version. Ideally, we would like for both to produce equivalent automata. If we can develop the technology to produce the same DFA from both these descriptions, it might make sense to write the longer RE since its correctness is far more obvious.

2.5 Implementing a DFA

DFAs are of interest to the compiler writer because they lead to efficient implementations. For example, we can implement the DFA for

r *digit digit**

with a straightforward code skeleton and a set of tables that encode the information contained in the drawing of the DFA. Remember that s_0 is the start state; that s_2 is the sole final state; and that s_e is an error state. We will assume the existence of a function $\mathcal{T}$: *state* → {*start,normal,final,error*} to let the recognizer switch into case logic based on the current state, and encode the transitions into a function δ : *state* × *character* → *state* that implements the transition function. These can both be encoded into tables.

```
char $\leftarrow$ next character;
state $\leftarrow$ $s_0$;
call action(char,state);

while ( char $\neq$ eof )
    state $\leftarrow$ $\delta$(state,char);
    call action(state, char);
    char $\leftarrow$ next character;

if $\mathcal{T}$[state] = final then
    report acceptance;
else
    report failure;
```

```
action(state,char)
    switch ($\mathcal{T}$[state])
        case start:
            word $\leftarrow$ char;
            break;
        case normal:
            word $\leftarrow$ word + char;
            break;
        case final:
            word $\leftarrow$ word + char;
            break;
        case error:
            print error message;
            break;
    end
```

Figure 2.3: A Skeleton Recognizer for "`r` *digit digit**"

δ	`r`	0,1,2,3,4 5,6,7,8,9	*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

$\mathcal{T}$	*action*
s_0	*start*
s_1	*normal*
s_2	*final*
s_e	*error*

Notice that we have compacted the tables to let them fit on the page. We denote this by listing multiple labels at the head of the column. This suggests the kind of compaction that can be achieved with relatively simple schemes. Of course, to use the compressed table, we must translate the input character into a table index with another table lookup.

The tables are interpreted to implement the recognizer; the code to accomplish this is shown in Figure 2.3. The code is quite simple. The code that precedes the *while* loop implements the recognizer's actions for state s_0. The other states are handled by the case statement in the routine *action*. (We pulled this out into a separate routine to make the structure of the skeleton parser more apparent.) The *while* loop iterates until all input is consumed; for each input character, it uses δ to select an appropriate transition, or next state. When it reaches the end of the input string, symbolized by *eof*, it determines if its last state is an accepting state by checking $\mathcal{T}$*[state]*.

To implement a second DFA, we can simply change the tables used with the skeleton recognizer. In the previous section, we presented the following refinement to the register specification.

$$\texttt{r}\ ((0 \mid 1 \mid 2)\ (\mathit{digit} \mid \epsilon)\ \mid\ (4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)\ \mid\ (3 \mid 30 \mid 31)$$

It restricts register names to the set `r0` through `r31`. The tables in Figure 2.4 encode the recognizer that we showed for this regular expression. Thus, to

δ	r	0,1,2	3	4-9	*other*
s_0	s_1	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_6	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e

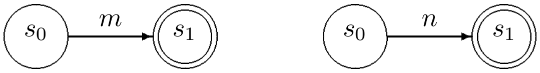
$\mathcal{T}$	*action*
s_0	*start*
s_1	*normal*
$s_{2,3,4,5,6}$	*final*
s_e	*error*

Figure 2.4: Recognizer Tables for the Refined Register Specification

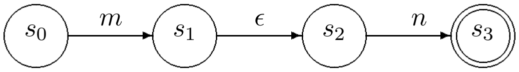
implement this tighter register specification, we need only to build a different set of tables for the skeleton recognizer.

2.6 Non-deterministic Finite Automata

Recall from the definition of a regular expression that we designated the empty string, ϵ as a regular expression. What role does ϵ play in a recognizer? We can use transitions on ϵ to combine FAs and form FAs for more complex regular expressions. For example, assume that we have FAs for the regular expressions m and n.

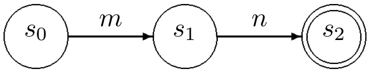


We can build an FA for mn by adding a transition on ϵ from the final state of FA$_m$ to the initial state of FA$_n$, renumbering the states, and using F_n as the set of final states for the new FA.



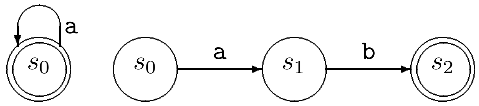
For this FA, the notion of acceptance changes slightly. In s_1, the FA takes a transition on ϵ (or no input) to s_2. Since ϵ is the empty string, it can occur between any two letters in the input stream without changing the word. This is a slight modification to the acceptance criterion, but seems intuitive.

By inspection, we can see that states s_1 and s_2 can be combined and the transition on ϵ eliminated.

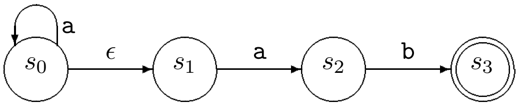


As we will see Section 2.7, this can be done systematically to eliminate all ϵ-transitions.

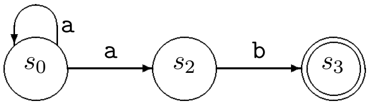
Sometimes, combining two DFAs with an ϵ-transition introduces complications into the transition function. Consider the following DFAs for the languages a^* and ab.



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



This FA has two possible transitions from state s_0 for an input of a. It can take the transition from s_0 back to s_0 labeled a. Alternatively, it can take the transition from s_0 to s_1 labeled ϵ and then take the transition from s_1 to s_2 labeled a. This problem is more clear if we coalesce the states connected by the ϵ-transition, s_0 and s_1.[2]



For the input string `aab`, the appropriate sequence of transitions is s_0, s_0, s_2, s_3. This consumes all of the input and leaves the FA in a final state. For the input string `ab`, however, the appropriate sequence of transitions is s_0, s_1, s_2. To accept these strings, the FA must select the transition out of s_0 that is appropriate for the context to the right of the current input character. Since the FA only has knowledge of the current state and the input character, it cannot know about context to the right.. This presents a fundamental dilemma that we will refer to as a *nondeterministic choice*. An FA that must make such a choice is called a *nondeterministic finite automata* (NFA). In contrast, a FA with unique character transitions in each state is a deterministic FA (DFA).

To make sense of this NFA, we need a new set of rules for interpreting its actions. Historically, two quite different explanations for the behavior of an NFA have been given.

- When the NFA confronts a nondeterministic choice, it always chooses the "correct" transition—that is, the transition that leads to accepting the input string, if any such transition exists. This model, using an omniscient

[2] In this case, we can safely coalesce s_0 and s_1. A general set of conditions for safely coalescing states is given in Section 2.9.1.

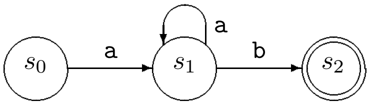
NFA, is appealing because it maintains (on the surface) the well-defined accepting mechanism of the NFA. Of course, implementing nondeterministic choice in this way would be quite difficult.

- When the NFA confronts a nondeterministic choice, it clones itself to pursue each possible transition. When any instance of the NFA exhausts its input and finds itself in a final state, it reports success and all its clones terminate. This model, while somewhat more complex, clearly pursues the complete set of paths through the NFA's transition graph. It correspond more closely to a realizable algorithm for interpreting the NFA's behavior.

In either model, it is worthwhile to formalize the acceptance criteria for an NFA. An NFA $(Q, \Sigma, \delta, q_0, F)$ halts on an input string $s_1 s_2 s_3 \ldots s_k$ if and only if there exists a path through the transition diagram that starts in q_0 and ends in some $q_k \in F$ such that the edge labels along the path spell out the input string. In other words, the i^{th} edge in the path must have the label s_i. This definition is consistent with either model of the NFA's behavior.

Any NFA can be simulated on a DFA. To see this intuitively, consider the set of states that represent a configuration of the NFA at some point in the input string. There can be, at most, a finite number of clones, each in a unique state.[3] We can build a DFA that simulates the NFA. States in the DFA will correspond to collections of states in the NFA. This may lead to an exponential blowup in the state space, since Q_{DFA} might be as large as $2^{Q_{NFA}}$. But, the resulting DFA still makes one transition per input character, so the simulation runs in time that grows linearly in the length of the input string. The NFA to DFA simulation has a potential space problem, but not a time problem. The actual construction is described in Section 2.7.

Note that the following DFA recognizes the same input language as our NFA for a^*ab

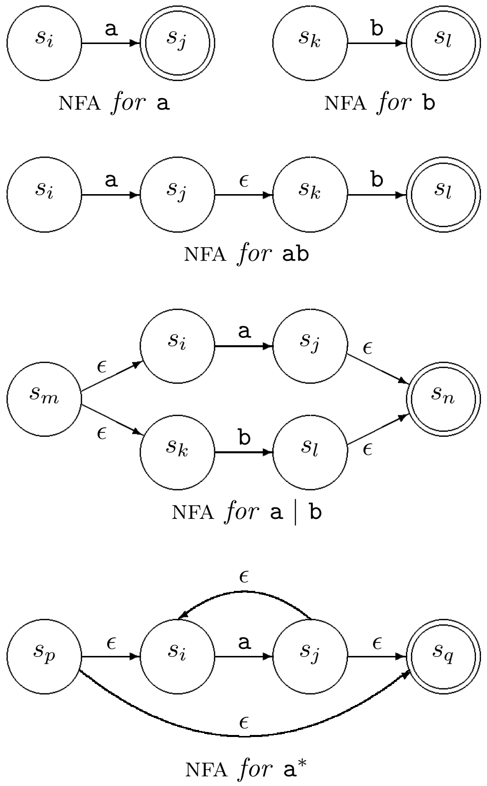


Rather than recognizing the language as a^*ab, it recognizes the equivalent language aa^*b, or a^+b.

More Closure Properties The relationship between a regular expression and a finite automaton provides a simple argument to show that regular expressions are closed under complement and intersection.

To see the closure property for complement, consider a regular expression r. To build a recognizer for the complement of r, written $\overline{r}$, we must first make the implicit transitions to an error state s_e explicit. This ensures that each state has a transition on every potential input symbol. Next, we reverse the designation

[3] The number of clones cannot exceed the length of the input string multiplied times the maximum number of nondeterministic transitions per state. Since both the input string and the transition graph are finite, their product must be finite.



NFA *for* a

NFA *for* b

NFA *for* ab

NFA *for* a | b

NFA *for* $\texttt{a}^*$

Figure 2.5: Components of Thompson's Construction

of final states—every final state becomes a non-final state and every non-final state becomes a final state. The resulting recognizer fails on every word in $L(r)$ and succeeds on every word not in $L(r)$. By definition, this automaton recognizes $L(\overline{r}) = \{w \in \Sigma^* \mid w \notin L(r)\}$.

Closure under complement is not strictly necessary for scanner construction. However, we can use this property to expand the range of input expressions allowed by a scanner generator. In some situations, complement provides a convenient notation for specifying a lexical pattern. Scanner generators often allow its use.

The argument for closure under intersection is somewhat more complex. It involves constructing an automaton whose state space contains the Cartesian product of the state spaces of the recognizers for r and s. The details are left as an exercise for the reader.

2.7 From Regular Expression to Scanner

The goal of our work with FAs is to automate the process of building an executable scanner from a collections of regular expressions. We will show how to accomplish this in two steps: using Thompson's construction to build an NFA from a RE [50] (Section 2.7.1) and using the subset construction to convert the NFA into a DFA (Section 2.7.2). The resulting DFA can be transformed into a minimal DFA—that is, one with the minimal number of states. That transformation is presented later, in Section 2.8.1.

In truth, we can construct a DFA directly from a RE. Since the direct construction combines the two separate steps, it may be more efficient. However, understanding the direct method requires a thorough knowledge of the individual steps. Therefore, we first present the two-step construction; the direct method is presented later, along with other useful transformations on automata.

2.7.1 Regular Expression to NFA

The first step in moving from a regular expression to an implemented scanner is deriving an NFA from the RE. The construction follows a straightforward idea. It has a template for building the NFA that corresponds to a single letter RE, and transformations on the NFAs to represent the impact of the RE operators, concatenation, alternation, and closure. Figure 2.5 shows the trivial NFA for the REs a and b, as well as the transformations to form the REs ab, a|b, and a*.

The construction proceeds by building a trivial NFA, and applying the transformations to the collection of trivial NFAs in the order of the relative precedence of the operators. For the regular expression a(b|c)*, the construction would proceed by building NFAs for a, b, and c. Next, it would build the NFA for b|c, then (b|c)*, and, finally, for a(b|c)*. Figure 2.6 shows this sequence of transformations.

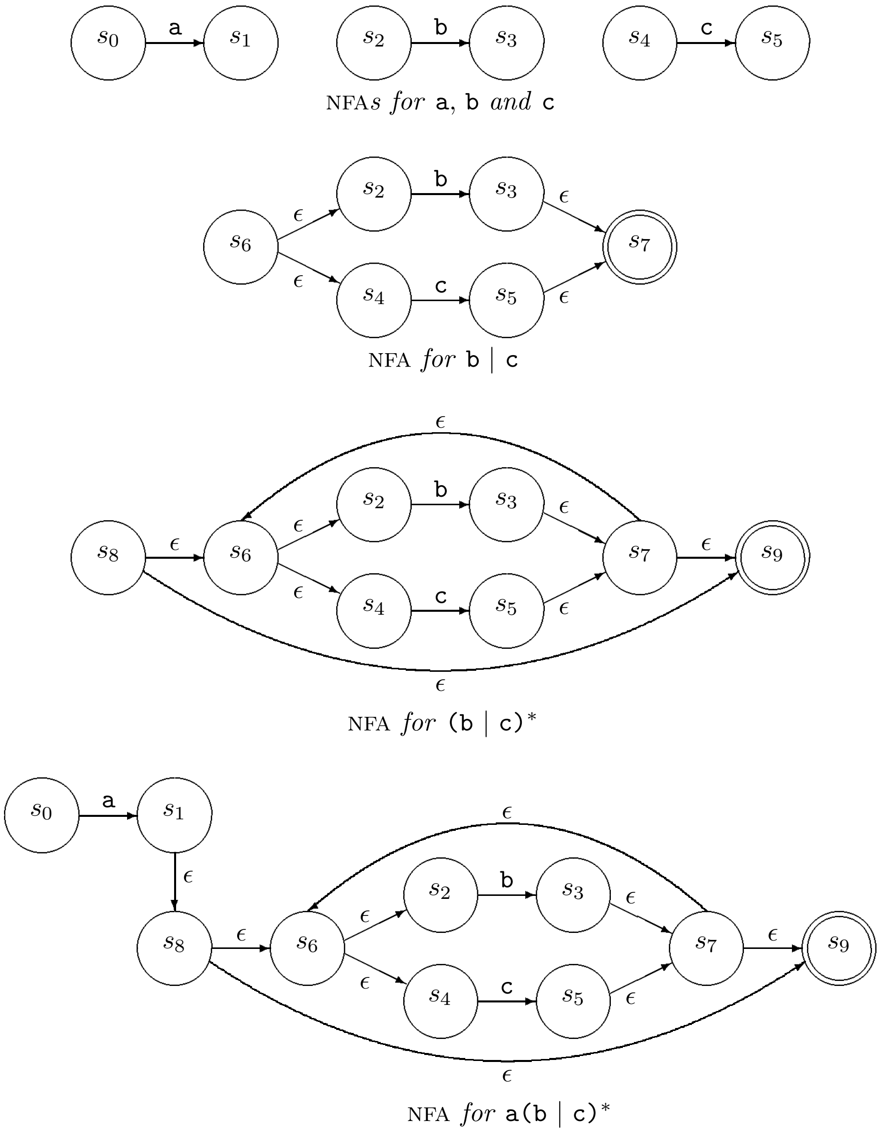
Thompson's construction relies on several properties of REs. It relies on the obvious and direct correspondence between the RE operators and the transformations on the NFAs. It combines this with the closure properties on REs for assurance that the transformations produce valid NFAs. Finally, it uses ϵ-moves to connect the subexpressions; this permits the transformations to be simple templates. For example, the template for a* looks somewhat contrived; it adds extra states to avoid introducing a cycle of ϵ-moves.

The NFAs derived from Thompson's construction have a number of useful properties.

1. Each has a single start state and a single final state. This simplifies the application of the transformations.

2. Any state that is the source or sink of an ϵ-move was, at some point in the process, the start state or final state of one of the NFAs representing a partial solution.

3. A 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.



NFAs *for* a, b *and* c

NFA *for* b | c

NFA *for* (b | c)*

NFA *for* a(b | c)*

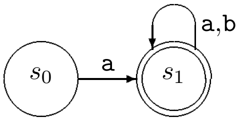
Figure 2.6: Thompson's construction for a(b|c)*

```
$S \leftarrow \epsilon$-closure($q0_N$)
while ($S$ is still changing)
    for each $s_i \in S$
        for each character $\alpha \in \Sigma$
            if $\epsilon$-closure(move($s_i$,$\alpha$) $\notin S$)
                add it to $S$ as $s_j$
                $T[s_i, \alpha] \leftarrow s_j$
```

Figure 2.7: The Subset Construction

These properties simplify an implementation of the construction. For example, instead of iterating over all the final states in the NFA for some arbitrary subexpression, the construction only needs to deal with a single final state.

Notice the large number of states in the NFA that Thompson's construction built for `a(b|c)*`. A human would likely produce a much simpler NFA, like the following:



We can automatically remove many of the ϵ-moves present in the NFA built by Thompson's construction. This can radically shrink the size of the NFA. Since the subset construction can handle ϵ-moves in a natural way, we will defer an algorithm for eliminating ϵ-moves until Section 2.9.1.

2.7.2 The Subset Construction

To construct a DFA from an NFA, we must build the DFA that simulates the behavior of the NFA on an arbitrary input stream. The process takes as input an NFA $N = (Q_N, \Sigma, \delta_N, q0_N, F_N)$ and produces a DFA $D = (Q_D, \Sigma, \delta_D, q0_D, F_D)$. The key step in the process is deriving Q_D and δ_D from Q_N and δ_N ($q0_D$ and F_D will fall out of the process in a natural way.) Figure 2.7 shows an algorithm that does this; it is often called the *subset construction*.

The algorithm builds a set S whose elements are themselves sets of states in Q_N. Thus, each $s_i \in S$ is itself a subset of Q_N. (We will denote the set of all subsets of Q_N as 2^{Q_N}, called the *powerset* of Q_N.) Each $s_i \in S$ represents a state in Q_D, so each state in Q_D represents a collection of states in Q_N (and, thus, is an element of 2^{Q_N}). To construct the initial state, $s_0 \in S$, it puts $q0_N$ into s_0 and then augments s_0 with every state in Q_N that can be reached from $q0_N$ by following one or more ϵ-transitions.

The algorithm abstracts this notion of following ϵ-transitions into a function, called *ϵ-closure* For a state, q_i, ϵ-*closure*(q_i) is the set containing q_i and any other states reachable from q_i by taking only ϵ-moves. Thus, the first step is to construct s_0 as *ϵ-closure($q0_N$)*.

Once S has been initialized with s_0, the algorithm repeatedly iterates over the elements of S, extending the partially constructed DFA (represented by S) by following transitions out of each $s_i \in S$. The *while* loop iterates until it completes a full iteration over S without adding a new set. To extend the partial DFA represented by S, it considers each s_i. For each symbol $\alpha \in \Sigma$, it collects together all the NFA states q_k that can be reached by a transition on α from a state $q_j \in s_i$.

In the algorithm, the computation of a new state is abstracted into a function call to *move*. *Move(s_i, α)* returns the set of states in 2^{Q_N} that are reachable from some $q_i \in Q_N$ by taking a transition on the symbol α. These NFA states form the core of a state in the DFA; we can call it s_j. To complete s_j, the algorithm takes its ϵ-*closure*. Having computed s_j, the algorithm checks if $s_j \in S$. If $s_j \notin S$, the algorithm adds it to S and records a transition from s_i to s_j on α.

The *while* loop repeats this exhaustive attempt to extend the partial DFA until an iteration adds no new states to S. The test in line 5 ensures that S contains no duplicate elements. Because each $s_i \in S$ is also an element of 2^{Q_n}, we know that this process must halt.

Sketch of Proof

1. 2^{Q_N} is finite. (It can be large, but is finite.)
2. S contains no duplicates.
3. The *while* loop adds elements to S; it cannot remove them.
4. S grows monotonically.

$\Rightarrow$ The loop halts.

When it halts, the algorithm has constructed model of the DFA that simulates Q_N. All that remains is to use S to construct Q_D and T to construct δ_D. Q_D gets a state q_i to represent each set $s_i \in S$; for any s_i that contains a final state of Q_N, the corresponding q_i is added to F_D, the set of final states for the DFA. Finally, the state constructed from s_0 becomes the initial state of the DFA.

Fixed Point Computations The subset construction is an example of a style of computation that arises regularly in Computer Science, and, in particular, in compiler construction. These problems are characterized by iterated application of a monotone function to some collection of sets drawn from a domain whose structure is known.[4] We call these techniques *fixed point* computations, because they terminate when they reach a point where further iteration produces the same answer—a "fixed point" in the space of successive iterates produced by the algorithm.

Termination arguments on fixed point algorithms usually depend on the known properties of the domain. In the case of the subset construction, we know that each $s_i \in S$ is also a member of 2^{Q_N}, the powerset of Q_N. Since Q_N is finite, 2^{Q_N} is also finite. The body of the *while* loop is monotone; it can only add elements to S. These facts, taken together, show that the *while* loop can execute only a finite number of iterations. In other words, it must halt because

[4] A function f is *monotone* if, $\forall x$ in its domain, $f(x) \geq x$.

```
$S \leftarrow$ $\epsilon$-closure($q_{0_N}$)
while ($\exists$ unmarked $s_i \in S$)
    mark $s_i$
    for each character $\alpha \in \Sigma$
        $t \leftarrow$ $\epsilon$-closure(move($s_i$,$\alpha$))
        if $t \notin S$ then
            add $t$ to $S$ as an unmarked state
        $T[s_i, \alpha] \leftarrow t$
```

Figure 2.8: A faster version of the Subset Construction

it can add at most $\mid 2^{Q_N} \mid$ elements to S; after that, it must halt. (It may, of course, halt much earlier.) Many fixed point computations have considerably tighter bounds, as we shall see.

Efficiency The algorithm shown in Figure 2.7 is particularly inefficient. It recomputes the transitions for each state in S on each iteration of the *while* loop. These transitions cannot change; they are wholly determined by the structure of the input NFA. We can reformulate the algorithm to capitalize on this fact; Figure 2.8 shows one way to accomplish this.

The algorithm in Figure 2.8 adds a "mark" to each element of S. When sets are added to S, they are unmarked. When the body of the *while* loop processes a set s_i, it marks s_i. This lets the algorithm avoid processing each s_i multiple times. It reduces the number of invocations of ϵ-*closure(move(*s_i,α*))* from $\mathbf{O}(\mid S \mid^2 \cdot \mid \Sigma \mid)$ to $\mathbf{O}(\mid S \mid \cdot \mid \Sigma \mid)$. Recall that S can be no larger than 2^{Q_N}.

Unfortunately, S can become rather large. The principal determinant of how much state expansion occurs is the degree of nondeterminism found in the input NFA. Recall, however, that the DFA makes exactly one transition per input character, independent of the size of Q_D. Thus, the use of non-determinism in specifying and building the NFA increases the space required to represent the corresponding DFA, but not the amount of time required for recognizing an input string.

Computing ϵ-closure as a Fixed Point To compute ϵ-*closure()*, we use one of two approaches: a straightforward, online algorithm that follows paths in the NFA's transition graph, or an offline algorithm that computes the ϵ-*closure* for each state in the NFA in a single fixed point computation.

```
for each state $n \in N$
    $E(n) \leftarrow \emptyset$
while (some $E(n)$ has changed)
    for each state $n \in N$
        $E(n) \leftarrow \bigcup_{\langle n,s,\epsilon \rangle} E(s)$
```

Here, we have used the notation $\langle n, s, \epsilon \rangle$ to name a transition from n to s on ϵ. Each $E(n)$ contains some subset of N (an element of 2^N). $E(n)$ grows monotonically since line five uses $\cup$ (not $\cap$). The algorithm halts when no $E(n)$ changes in an iteration of the outer loop. When it halts, $E(n)$ contains the names of all states in *ϵ-closure(n)*.

We can obtain a tighter time bound by observing that $\mid E(n) \mid$ can be no larger than the number of states involved in a path leaving n that is labeled entirely with ϵ's. Thus, the time required for a computation must be related to the number of nodes in that path. The largest $E(n)$ set can have N nodes. Consider that longest path. The algorithm cannot halt until the name of the last node on the path reaches the first node on the path. In each iteration of the outer loop, the name of the last node must move one or more steps closer to the head of the path. Even with the worst ordering for that path, it must move along one edge in the path.

At the start of the iteration, $n_{last} \in E(n_i)$ for some n_i. If it has not yet reached the head of the path, then there must be an edge $\langle n_i, n_j, \epsilon \rangle$ in the path. That node will be visited in the loop at line six, so n_{last} will move from $E(n_i)$ to $E(n_j)$. Fortuitous ordering can move it along more than one ϵ-transition in a single iteration of the loop at line six, but it must always move along at least one ϵ-transition, unless it is in the last iteration of the outer loop.

Thus, the algorithm requires at most one while loop iteration for each edge in the longest ϵ-path in the graph, plus an extra iteration to recognize that the E sets have stabilized. Each iteration visits N nodes and does E unions. Thus, its complexity is $\mathsf{O}(N(N+E))$ or $\mathsf{O}(\max(N^2, NE))$. This is much better than $\mathsf{O}(2^N)$.

We can reformulate the algorithm to improve its specific behavior by using a worklist technique rather than a round-robin technique.

```
for each state $n \in N$
    E(n) $\leftarrow \emptyset$
WorkList $\leftarrow N$
while (WorkList $\neq \emptyset$ )
    remove $n_i$ from worklist
    E($n_j$) $\leftarrow \bigcup_{\langle n_i, n_j, \epsilon \rangle}$ E($n_j$)
    if E($n_j$) changed then
        WorkList $\leftarrow$ WorkList $\cup \{n_k \mid \langle n_k, n_i, \epsilon \rangle \in \delta_{NFA}\}$
```

This version only visits a node when the E set at one of its ϵ-successors has changed. Thus, it may perform fewer union operations than the round robin version. However, its asymptotic behavior is the same. The only way to improve its asymptotic behavior is to change the order in which nodes are removed from the worklist. This issue will be explored in some depth when we encounter data-flow analysis in Chapter 13.

2.7.3 Some final points

Thus far, we have developed the mechanisms to construct a DFA implementation from a single regular expression. To be useful, a compiler's scanner must recognize all the syntactic categories that appear in the grammar for the source language. What we need, then, is a recognizer that can handle all the REs for the language's micro-syntax. Given the REs for the various syntactic categories, $r_1, r_2, r_3, \ldots, r_k$, we can construct a single RE for the entire collection by forming $(r_1 \mid r_2 \mid r_3 \mid \ldots \mid r_k)$.

If we run this RE through the entire process, building NFAs for the subexpressions, joining them with ϵ-transitions, coalescing states, constructing the DFA that simulates the NFA, and turning the DFA into executable code, we get a scanner that recognizes precisely one word. That is, when we invoke it on some input, it will run through the characters one at a time and accept the string if it is in a final state when it exhausts the input. Unfortunately, most real programs contain more than one word. We need to transform either the language or the recognizer.

At the language level, we can insist that each word end with some easily recognizable delimiter, like a blank or a tab. This is deceptively attractive. Taken literally, it would require delimiters surrounding commas, operators such as + and -, and parentheses.

At the recognizer level, we can transform the DFA slightly and change the notion of accepting a string. For each final state, q_i, we (1) create a new state q_j, (2) remove q_i from F and add q_j to F, and (3) make the error transition from q_i go to q_j. When the scanner reaches q_i and cannot legally extend the current word, it will take the transition to q_j, a final state. As a final issue, we must make the scanner stop, backspace the input by one character, and accept in each new final state. With these modifications, the recognizer will discover the longest legal keyword that is a prefix of the input string.

What about words that match more than one pattern? Because the methods described in this chapter build from a base of non-determinism, we can union together these arbitrary REs without worrying about conflicting rules. For example, the specification for an Algol identifier admits all of the reserved keywords of the language. The compiler writer has a choice on handling this situation. The scanner can recognize those keywords as identifiers and look up each identifier in a pre-computed table to discover keywords, or it can include a RE for each keyword. This latter case introduces non-determinism; the transformations will handle it correctly. It also introduces a more subtle problem—the final NFA reaches two distinct final states, one recognizing the keyword and the other recognizing the identifier, and is expected to consistently choose the former. To achieve the desired behavior, scanner generators usually offer a mechanism for prioritizing REs to resolve such conflicts.

LEX and its descendants prioritize patterns by the order in which they appear in the input file. Thus, placing keyword patterns before the identifier pattern would ensure the desired behavior. The implementation can ensure that the final states for patterns are numbered in a order that corresponds to this priority

```
P ← { F, (Q - F) }
while (P is still changing)
    T ← $\emptyset$
    for each set $s \in$ P
        for each $\alpha \in \Sigma$
            partition $s$ by $\alpha$
                into $s_1, s_2, s_3, \ldots s_k$
            T ← T $\cup$ $s_1, s_2, s_3, \ldots s_k$
    if T $\neq$ P then
        P ← T
```

Figure 2.9: DFA minimization algorithm

ordering. When the scanner reaches a state representing multiple final states, it uses the action associated with the lowest-numbered final state.

2.8 Better Implementations

A straightforward scanner generator would take as input a set of regular expressions, construct the NFA for each RE, combine them using ϵ-transitions (using the pattern for a|b in Thompson's construction), and perform the subset construction to create the corresponding DFA. To convert the DFA into an executable program, it would encode the transition function into a table indexed by current state and input character, and plug the table into a fairly standard skeleton scanner, like the one shown in Figure 2.3.

While this path from a collection of regular expressions to a working scanner is a little long, each of the steps is well understood. This is a good example of the kind of tedious process that is well suited to automation by a computer. However, a number of refinements to the automatic construction process can improve the quality of the resulting scanner or speed up the construction.

2.8.1 DFA minimization

The NFA to DFA conversion can create a DFA with a large set of states. While this does not increase the number of instructions required to scan a given string, it does increase the memory requirements of the recognizer. On modern computers, the speed of memory accesses often governs the speed of computation. Smaller tables use less space on disk, in RAM, and in the processor's cache. Each of those can be an advantage.

To minimize the size of the DFA, $D = (Q, \Sigma, \delta, q_0, F)$, we need a technique for recognizing when two states are equivalent—that is, they produce the same behavior on any input string. Figure 2.9 shows an algorithm that partitions the states of a DFA into equivalence classes based on their behavior relative to an input string.

Because the algorithm must also preserve halting behavior, the algorithm cannot place a final state in the same class as a non-final state. Thus, the initial partitioning step divides Q into two equivalence classes, F and $Q - F$.

Each iteration of the *while* loop refines the current partition, P, by splitting apart sets in P based on their outbound transitions. Consider a set $p = \{q_i, q_j, q_k\}$ in the current partition. Assume that q_i, q_j, and q_k all have transitions on some symbol $\alpha \in \Sigma$, with $q_x = \delta(q_i, \alpha)$, $q_y = \delta(q_j, \alpha)$, and $q_z = \delta(q_k, \alpha)$. If all of q_x, q_y, and q_z are in the same set in the current partition, then q_i, q_j, and q_k should remain in the same set in the new partition. If, on the other hand, q_z is in a different set than q_x and q_y, then the algorithm splits p into $p_1 = \{q_i, q_j\}$ and $p_2 = \{q_k\}$, and puts both p_1 and p_2 into the new partition. This is the critical step in the algorithm.

When the algorithm halts, the final partition cannot be refined. Thus, for a set $s \in P$, the states in s cannot be distinguished by their behavior on an input string. From the partition, we can construct a new DFA by using a single state to represent each set of states in P, and adding the appropriate transitions between these new representative states. For each state $s \in P$, the transition out of s on some $\alpha \in \Sigma$ must go to a single set t in P; if this were not the case, the algorithm would have split s into two or more smaller sets.

To construct the new DFA, we simply create a state to represent each $p \in P$, and add the appropriate transitions. After that, we need to remove any states not reachable from the entry state, along with any state that has transitions back to itself on every $\alpha \in \Sigma$. (Unless, of course, we want an explicit representation of the error state.) The resulting DFA is minimal; we leave the proof to the interested reader.

This algorithm is another example of a fixed point computation. P is finite; at most, it can contain $\mid Q \mid$ elements. The body of the *while* loop can only increase the size of P; it splits sets in P but never combines them. The worst case behavior occurs when each state in Q has different behavior; in that case, the *while* loop halts when P has a unique set for each $q \in Q$. (This would occur if the algorithm was invoked on a minimal DFA.)

2.8.2 Programming Tricks

Explicit State Manipulation Versus Table Lookup The example code in Figure 2.3 uses an explicit variable, *state*, to hold the current state of the DFA. The *while* loop tests *char* against *eof*, computes a new state, calls *action* to interpret it, advances the input stream, and branches back to the top of the loop. The implementation spends much of its time manipulating or testing the state (and we have not yet explicitly discussed the expense incurred in the array lookup to implement the transition table or the logic required to support the *switch* statement (see Chapter 8).

We can avoid much of this overhead by encoding the state information implicitly in the program counter. In this model, each state checks the next character against its transitions, and branches directly to the next state. This creates a program with complex control flow; it resembles nothing as much as a jumbled

```
        char ← next character;

$s_0$:  word ← char ;
        char ← next character;
        if (char = 'r') then
            goto $s_1$;
        else goto $s_e$;

$s_1$:  word ← word + char;
        char ← next character;
        if ('0'≤char≤'9') then
            goto $s_2$;
        else goto $s_e$;

$s_2$:  word ← word + char;
        char ← next character;
        if ('0'≤char≤'9') then
            goto $s_2$;
        else if (char = eof) then
            report acceptance;
        else goto $s_e$;

$s_e$:  print error message;
        return failure;
```

Figure 2.10: A direct-coded recognizer for "r *digit digit**"

heap of spaghetti. Figure 2.10 shows a version of the skeleton recognizer written in this style. It is both shorter and simpler than the table-driven version. It should be faster, because the overhead per state is lower than in table-lookup version.

Of course, this implementation paradigm violates many of the precepts of structured programming. In a small code, like the example, this style may be comprehensible. As the RE specification becomes more complex and generates both more states and more transitions, the added complexity can make it quite difficult to follow. If the code is generated directly from a collection of REs, using automatic tools, there is little reason for a human to directly read or debug the scanner code. The additional speed obtained from lower overhead and better memory locality[5] makes direct-coding an attractive option.

Hashing Keywords versus Directly Encoding Them The scanner writer must choose how to specify reserved keywords in the source programming language—words like `for`, `while`, `if`, `then`, and `else`. These words can be written as regular expressions in the scanner specification, or they can be folded into the set of identifiers and recognized using a table lookup in the actions associated with an identifier.

With a reasonably implemented hash table, the expected case behavior of the two schemes should differ by a constant amount. The DFA requires time proportional to the length of the keyword, and the hash mechanism adds a constant time overhead after recognition.

From an implementation perspective, however, direct coding is simpler. It avoids the need for a separate hash table of reserved words, along with the cost of a hash lookup on every identifier. Direct coding increases the size of the DFA from which the scanner is built. This can make the scanner's memory requirements larger and might require more code to select the transitions out

[5] Large tables may have rather poor locality.

of some states. (The actual impact of these effects undoubtedly depend on the behavior of the memory hierarchy.)

On the other hand, using a reserved word table also requires both memory and code. With a reserved word table, the cost of recognizing every identifier increases.

Specifying Actions In building a scanner generator, the designer can allow actions on each transition in the DFA or only in the final states of the DFA. This choice has a strong impact on the efficiency of the resulting DFA. Consider, for example, a RE that recognizes positive integers with a single leading zero.

$$0 \mid (1 \mid 2 \mid \cdots \mid 9)(0 \mid 1 \mid 2 \cdots \mid 9)^*$$

An scanner generator that allows actions only in accepting states will force the user to rescan the string to compute its actual value. Thus, the scanner will step through each character of the already recognized word, performing some appropriate action to convert the text into a decimal value. Worse yet, if the system provides a built-in mechanism for the conversion, the programmer will likely use it, adding the overhead of a procedure call to this simple and frequently executed operation. (On Unix systems, many LEX-generated scanners contain an action that invokes `sscanf()` to perform precisely this function.)

If, however, the scanner generator allows actions on each transition, the compiler writer can implement the ancient assembly-language trick for this conversion. On recognizing an initial digit, the accumulator is set to the value of the recognized digit. On each subsequent digit, the accumulator is multiplied by ten and the new digit added to it. This algorithm avoids touching the character twice; it produces the result quickly and inline using the well-known conversion algorithm; and it eliminates the string manipulation overhead implicit in the first solution. (The scanner likely copies characters from the input buffer into some result string before on each transition in the first scenario.)

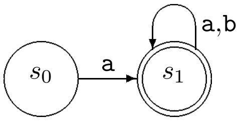
In general, the scanner should avoid processing each character multiple times. The more freedom that it allows the compiler writer in the placement of actions, the simpler it becomes to implement effective and efficient algorithms that avoid copying characters around and examining them several times.

2.9 Related Results

Regular expressions and their corresponding automata have been studied for many years. This section explores several related issues. These results do not play a direct role in scanner construction; however, they may be of intellectual interest in the discussion of scanner construction.

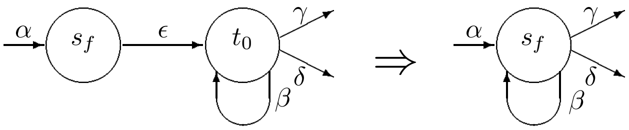
2.9.1 Eliminating ϵ-moves

When we applied Thompson's construction to the regular expression `a(b|c)`*, the resulting NFA had ten states and twelve transitions. All but three of the transitions are ϵ-moves. A typical compiler-construction student would produce a two state DFA with three transitions and no ϵ-moves.

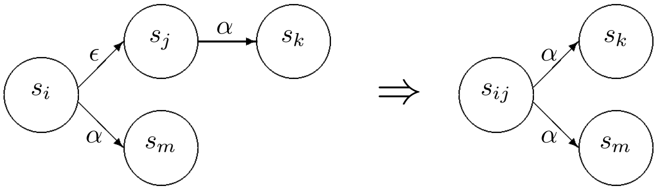


Eliminating ϵ-moves can both shrink and simplify an NFA. While it is not strictly necessary in the process of converting a set of REs into a DFA, it can be helpful if humans are to examine the automata at any point in the process.

Some ϵ-moves can be easily eliminated. The NFA shown on the left can arise from Thompson's construction. The source of the ϵ-move, state s_f, was a final state for some subexpression of the RE ending in α; the sink of the ϵ-move, state t_0, was the initial state of another subexpression beginning with either γ, δ, or θ.

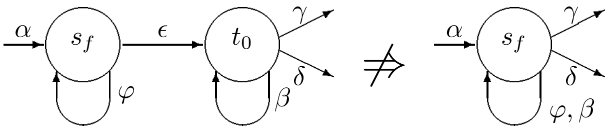


In this particular case, we can eliminate the ϵ-move by combining the two states, s_f and t_0, into a single state. To accomplish this, we need to make s_f the source of each edge leaving t_0, and s_f the sink of any edge entering t_0. This produces the simplified NFA shown on the right. Notice that coalescing can create a state with multiple transitions on the same symbol:



If s_k and s_m are distinct states, then both $\langle s_{ij}, s_k, \alpha\rangle$ and $\langle s_{ij}, s_m, \alpha\rangle$ should remain. If s_k and s_m are the same state, then a single transition will suffice. A more general version of this problem arises if s_k and s_m are distinct, but the sub-NFAs that they begin recognize the same languages. The DFA minimization algorithm should eliminate this latter kind of duplication.

Some ϵ-moves can be eliminated by simply coalescing states. Obviously, it works when the ϵ-move is the only edge leaving its source state and the only edge entering its sink state. In general, however, the states connected by an ϵ-move cannot be directly coalesced. Consider the following modification of the earlier NFA, where we have added one additional edge—a transition from s_f to itself on φ.



```
for each edge $e \in E$
    if $e = \langle q_i, q_j, \epsilon \rangle$ then
        add $e$ to WorkList

while (WorkList $\neq \emptyset$)
    remove $e = \langle q_i, q_j, \alpha \rangle$ from WorkList
    if $\alpha = \epsilon$ then
        for each $\langle q_j, q_k, \beta \rangle$ in $E$
            add $\langle q_i, q_k, \beta \rangle$ to $E$
            if $\beta = \epsilon$ then
                add $\langle q_i, q_k, \beta \rangle$ to WorkList
        delete $\langle q_i, q_j, \epsilon \rangle$ from $E$
        if $q_j \in F$ then
            add $q_i$ to $F$

for each state $q_i \in N$
    if $q_i$ has no entering edge then
        delete $q_i$ from $N$
```

Figure 2.11: Removing ϵ-transitions

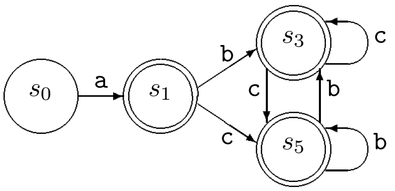
The NFA on the right would result from combining the states. Where the original NFA accepted words that contained the substring $\varphi^*\beta^*$, the new NFA accepts words containing $(\varphi \mid \beta)^*$. Coalescing the states changed the language!

Figure 2.11 shows an algorithm that eliminates ϵ-moves by duplicating transitions. The underlying idea is quite simple. If there exists a transition $\langle q_i, q_j, \epsilon \rangle$, it copies each transition leaving q_j so that an equivalent transition leaves q_i, and then deletes $\langle q_i, q_j, \epsilon \rangle$. This has the effect of eliminating paths of the form $\epsilon^*\alpha$ and replacing them with a direct transition on α.

To understand its behavior, let's apply it to the NFA for a(b|c)* shown in Figure 2.6. The first step puts all of the ϵ-moves onto a worklist. Next, the algorithm iterates over the worklist, copying edges, deleting ϵ-moves, and updating the set of final states, F. Figure 2.12 summarizes the iterations. The left column shows the edge removed from the worklist; the center column shows the transitions added by copying; the right column shows any states added to F. To clarify the algorithm's behavior, we have removed edges from the worklist in phases. The horizontal lines divide the table into phases. Thus, the first section, from $\langle$1,8,$\epsilon\rangle$ to $\langle$7,9,$\epsilon\rangle$ contains all the edges put on the worklist initially. The second section includes all edges added during the first phase. The final section includes all edges added during the second phase. Since it adds no additional ϵ-moves, the worklist is empty and the algorithm halts.

ϵ-move from		*Add*
WorkList	*Adds transitions*	*to F*
1,8	$\langle 1,9,\epsilon \rangle$ $\langle 1,6,\epsilon \rangle$	
8,6	$\langle 8,2,\epsilon \rangle$ $\langle 8,4,\epsilon \rangle$	
8,9	—	8
6,2	⟨6,3,b⟩	
6,4	⟨6,5,c⟩	
3,7	$\langle 3,6,\epsilon \rangle$ $\langle 3,9,\epsilon \rangle$	3
5,7	$\langle 5,6,\epsilon \rangle$ $\langle 5,9,\epsilon \rangle$	5
7,6	$\langle 7,2,\epsilon \rangle$ $\langle 7,4,\epsilon \rangle$	
7,9	—	7
1,9	—	1
1,6	⟨1,3,b⟩ ⟨1,5,c⟩	
8,2	⟨8,3,b⟩	
8,4	⟨8,5,c⟩	
3,6	$\langle 3,2,\epsilon \rangle$ $\langle 3,4,\epsilon \rangle$	
3,9	—	3
5,6	$\langle 5,2,\epsilon \rangle$ $\langle 5,4,\epsilon \rangle$	
5,9	—	5
7,2	⟨7,3,b⟩	
7,4	⟨7,5,c⟩	
3,2	⟨3,3,b⟩	
3,4	⟨3,5,c⟩	
5,2	⟨5,3,b⟩	
5,4	⟨5,5,c⟩	

Figure 2.12: ϵ-removal algorithm applied to a(b|c)*



The resulting NFA is much simpler than the original. It has four states and seven transitions, none on ϵ. Of course, it is still somewhat more complex than the two state, three transition NFA shown earlier. Applying the DFA minimization algorithm would simplify this automaton further.

2.9.2 Building a RE from a DFA

In Section 2.7, we showed how to build a DFA from an arbitrary regular expression. This can be viewed as a constructive proof that DFAs are at least as powerful as REs. In this section, we present a simple algorithm that constructs

```
for i = 1 to N
    for j = 1 to N
        $R^0_{ij} = \{a \mid \delta(s_i, a) = s_j\}$
        if (i = j) then
            $R^0_{ij} = R^0_{ij} \cup \{\epsilon\}$
for k = 1 to N
    for i = 1 to N
        for j = 1 to N
            $R^k_{ij} = R^{k-1}_{ik}(R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1}_{ij}$
L = $\bigcup_{s_j \in F} R^N_{1j}$
```

Figure 2.13: From a DFA to a RE

a RE to describe the set of strings accepted by an arbitrary DFA. It shows that REs are at least as powerful as DFAs. Taken together, these constructions form the basis of a proof that REs are equivalent to DFAs.

Consider the diagram of a DFA as a graph with labeled edges. The problem of deriving a 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 edge labels for every path from q_0 to q_i, $\forall q_i \in F$. For any DFA with a cyclic transition graph, the set of such paths is infinite. Fortunately, the REs have the Kleene-closure operator to handle this case and summarize the complete set of sub-paths created by a cycle.

Several techniques can be used to compute this path expression. The algorithm generates an expression that represents the labels along all paths between two nodes, for each pair of nodes in the transition diagram. Then, it unions together the expressions for paths from q_0 to q_i, $\forall q_i \in F$. This algorithm, shown in Figure 2.13, systematically constructs the path expressions for all paths. Assume, without loss of generality, that we can number the nodes from 1 to N, with q_0 having the number 1.

The algorithm computes a set of expressions, denoted R^k_{ij}, for all the relevant values of i, j, and k. R^k_{ij} is an expression that describes all paths through the transition graph, from state i to state j without going through a state numbered higher than k. Here, "through" means both entering and leaving, so that $R^2_{1,16}$ can be non-empty.

Initially, it sets R^0_{ij} to contain the labels of all edges that run directly from i to j. Over successive iterations, it builds up longer paths by adding to R^{k-1}_{ij} the paths that actually pass through k on their way from i to j. Given R^{k-1}_{ij}, the set of paths added by going from $k-1$ to k is exactly the set of paths that run from i to j using no state higher than $k-1$, concatenated with the paths from k to itself that pass through no state higher than $k-1$, followed by the paths from k to j that pass through no state higher than $k-1$. That is, each iteration of the loop on k adds the paths that pass through k to each set R^{k-1}_{ij}.

```
 1.        INTEGERFUNCTIONA
 2.        PARAMETER(A=6,B=2)
 3.        IMPLICIT CHARACTER*(A-B)(A-B)
 4.        INTEGER FORMAT(10),IF(10),DO9E1
 5.    100 FORMAT(4H)=(3)
 6.    200 FORMAT(4 )=(3)
 7.        DO9E1=1
 8.        DO9E1=1,2
 9.      9 IF(X)=1
10.        IF(X)H=1
11.        IF(X)300,200
12.    300 END
13.  C     this is a comment
14.       $FILE(1)
15.        END
```

Figure 2.14: Scanning FORTRAN

When the k-loop terminates, the various R^k_{ij} expressions account for all paths in the transition graph. Now, we must compute the set of paths that being in state 1 and end in some final state, $s_j \in F$.

2.10 Lexical Follies of Real Programming languages

This chapter has dealt, largely, with the theory of specifying and automatically generating scanners. Most modern programming languages have a simple lexical structure. In fact, the development of a sound theoretical basis for scanning probably influenced language design in a positive way. Nonetheless, lexical difficulties do arise in the design of programming languages. This section presents several examples.

To see how difficult scanning can be, consider the example FORTRAN fragment shown in Figure 2.14. (The example is due to Dr. F.K. Zadeck.) In FORTRAN 66 (and FORTRAN 77), blanks are not significant—the scanner ignores them. Identifiers are limited to six characters, and the language relies on this property to make some constructs recognizable.

In line 1, we find a declaration of `A` as an integer function. To break this into words, the scanner must read `INTEGE` and notice that the next character, `R`, is neither an open parenthesis, as in `INTEGE(10) = J` nor an assignment operator, as in `INTEGE = J` This fact, combined with the six character limit on identifiers, lets the scanner understand that `INTEGE` is the start of the reserved keyword `INTEGER`. The next reserved keyword, `FUNCTION` requires application of the same six character limit. After recognizing that `FUNCTIONA` has too many characters to be an `INTEGER` variable, the scanner can conclude that it has three words on the first line, `INTEGER`, `FUNCTION`, and `A`.

The second line declares `A` as a `PARAMETER` that is macro-expanded to `6` when

Digression: The Hazards of Bad Lexical Design

An apocryphal story has long circulated in the compiler construction community. It suggests than an early NASA mission to Mars (or Venus, or the Moon,) crashed because of a missing comma in the FORTRAN code for a DO loop. Of course, the body of the loop would have executed once, rather than the intended number of times. While we doubt the truth of the story, it has achieved the status of an "urban legend" among compiler writers.

it occurs as a word on its own. Similarly, `B` expands to `2`. Again, the scanner must rely on the six character limit.

With the parameters expanded, line three scans as `IMPLICIT CHARACTER*4 (A-B)`. It tells the compiler that any variable beginning with the letters `A` or `B` has the data type of a four-character string. Of course, the six character limit makes it possible for the scanner to recognize `IMPLICIT`.

Line four has no new lexical complexity. It declares `INTEGER` arrays of ten element named `FORMAT` and `IF`, and a scalar `INTEGER` variable named `DO9E1`.

Line five begins with a statement label, `100`. Since FORTRAN was designed for punch cards, it has a fixed-field format for each line. Columns 1 through 5 are reserved for statement labels; a `C` in column 1 indicates that the entire line is a comment. Column 6 is empty, unless the line is a continuation of the previous line. The remainder of line five is a `FORMAT` statement. The notation `4H)=(3` is a "Hollerith constant." `4H` indicates that the following four characters form a literal constant. Thus, the entire line scans as:

⟨*label*, `100`⟩, ⟨*format keyword*⟩, ⟨'('⟩ ⟨*constant*, `‘‘)=(3’’`⟩ ⟨')'⟩

This is a `FORMAT` statement, used to specify the way that characters are read or written in a `READ`, `WRITE`, or `PRINT` statement.

Line six is an assignment of the value `3` to the fourth element of the `INTEGER` array `FORMAT`, declared back on line 4. To distinguish between the variable and the keyword, the scanner must read past the `(4 )` to reach the equals sign. Since the equals sign indicates an assignment statement, the text to its left must be a reference to a variable. Thus, `FORMAT` is the variable rather than the keyword. Of course, adding the `H` from line 5 would change that interpretation.

Line 7 assigns the value 1 to the variable `DO9E1`, while line 8 marks the beginning of a `DO` loop that ends at label `9` and uses the induction variable `E1`. The difference between these lines lies in the comma following `=1`. The scanner cannot decide whether `DO9E1` is a variable or the sequence ⟨*keyword*,`DO`⟩, ⟨*label*,`9`⟩, ⟨*variable*,`E1`⟩ until it reaches either the comma or the end of the line.

The next three lines look quite similar, but scan differently. The first is an assignment of the value `1` to the $\texttt{X}^{th}$ element of the array `IF` declared on line 4. The second is an `IF` statement that assigns `1` to `H` if `X` evaluates to true. The third branches to either label `200` or `300`, depending on the relationship between the value of `X` and zero. In each case, the scanner must proceed well beyond the `IF` before it can classify `IF` as a variable or a keyword.

The final complication begins on line 12. Taken by itself, the line appears to be an `END` statement, which usually appears at the end of a procedure. It is followed, on line 13, by a comment. The comment is trivially recognized by of the character in column 1. However, line 14 is a continuation of the previous statement, on line 12. To see this, the scanner must read line 13, discover that it is a comment, and read line 13 to discover the `$` in column 6. At this point, it finds the string `FILE`. Since blanks (and intervening comment cards) are not significant, the word on line 12 is actually `ENDFILE`, split across an internal comment. Thus, lines 12 and 14 form an `ENDFILE` statement that marks the file designated as `1` as finished.

The last line, 15, is truly an `END` statement.

To scan this simple piece of Fortran text, the scanner needed to look arbitrarily far ahead in the text—limited only by the end of the statement. In the process, it applied idiosyncratic rules related to identifier length, to the placement of symbols like commas and equal signs. It had to read to the end of some statements to categorize the initial word of a line.

While these problems in Fortran are the result of language design from the late 1950's, more modern languages have their own occasional lexical lapses. For example, pl/i, designed a decade later, discarded the notion of reserved keywords. Thus, the programmer could use words like `if`, `then`, and `while` as variable names. Rampant and tasteless use of that "feature" led to several examples of lexical confusion.

```
if then then then = else; else else = then;
```

This code fragment is an `if-then-else` construct that controls assignments between two variables named `then` and `else`. The choice between the `then`-part and the `else`-part is based on an expression consisting of a single reference to the variable `then`. It is unclear why anyone would want to write this fragment.

More difficult, from a lexical perspective, is the following set of statements.

```
declare (a1,a2,a3,a4) fixed binary;
declare (a1,a2,a3,a4) = 2;
```

The first declares four integer variables, named `a1`, `a2`, `a3`, and `a4`. The second is an assignment to an element of a four-dimensional array named `declare`. (It presupposes the existence of a declaration for the array.) This example exhibits a Fortran-like problem. The compiler must scan to `=` before discovering whether `declare` is a keyword or an identifier. Since pl/i places no limit on the comma-separated list's length, the scanner must examine an arbitrary amount of right context before it can determine the syntactic category for `declare`. This complicates the problem of buffering the input in the scanner.

As a final example, consider the syntax of c++, a language designed in the late 1980s. The template syntax of c++ allows the fragment

```
PriorityQueue<MyType>
```

If `MyType` is itself a template, this can lead to the fragment

```
PriorityQueue<MyType<int>>
```

which seems straight forward to scan. Unfortunately, `>>` is a C++ operator for writing to the output stream, making this fragment mildly confusing. The C++ standard actually requires one or more blank between two consecutive angle brackets that end a template definition. However, many C++ compilers recognize this detail as one that programmers will routinely overlook. Thus, they correctly handle the case of the missing blank. This confusion can be resolved in the parser by matching the angled brackets with the corresponding opening brackets. The scanner, of course, cannot match the brackets. Recognizing `>>` as either two closing occurrences of `>` or as a single operator requires some coordination between the scanner and the parser.

2.11 Summary and Perspective

The widespread use of regular expressions 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 compilers as a means of concisely specifying groups of strings (that happen to be regular languages).

Most modern compilers use generated scanners. The properties of deterministic finite automata match quite closely the demands of a compiler. The cost of recognizing a word is proportional to its length. The overhead per character is quite small in a careful implementation. The number of states can be reduced with the widely-used minimization algorithm. Direct-encoding of the states provides a speed boost over a table-driven interpreter. The widely available scanner generators are good enough that hand-implementation can rarely, if ever, be justified.

Questions

1. Consider the following regular expression:

   $$\texttt{r0} \mid \texttt{r00} \mid \texttt{r1} \mid \texttt{r01} \mid \texttt{r2} \mid \texttt{r02} \mid \quad \ldots \quad \mid \texttt{r30} \mid \texttt{r31}$$

   Apply the constructions to build

   (a) the NFA from the RE,

   (b) the DFA from the NFA, and

   (c) the RE from the DFA.

   Explain any differences between the original RE and the RE that you produced.

   How does the DFA that you built compare with the DFA built in the chapter from following RE

   $$\texttt{r}\ ((0 \mid 1 \mid 2)\ (\mathit{digit} \mid \epsilon)\ \mid\ (4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)\ \mid\ (3 \mid 30 \mid 31)\ ?$$

Chapter 3

Parsing

3.1 Introduction

The parser's task is to analyze the input program, as abstracted by the scanner, and determine whether or not the program constitutes a legal sentence in the source language. Like lexical analysis, syntax analysis has been studied extensively. As we shall see, results from the formal treatment of syntax analysis lead to the creation of efficient parsers for large families of languages.

Many techniques have been proposed for parsing. Many tools have been built that largely automate parser construction. In this chapter, we will examine two specific parsing techniques. Both techniques are capable of producing robust, efficient parsers for typical programming languages. Using the first method, called *top-down, recursive-descent parsing*, we will construct a hand-coded parser in a systematic way. Recursive-descent parsers are typically compact and efficient. The parsing algorithm used is easy to understand and implement. The second method, called *bottom-up, LR(1) parsing*, uses results from formal language theory to construct a parsing automaton. We will explore how tools can directly generate a parsing automaton and its implementation from a specification of the language's syntax. LR(1) parsers are efficient and general; the tools for building LR(1) parsers are widely available for little or no cost.

Many other techniques for building parsers have been explored in practice, in the research literature, and in other textbooks. These include bottom-up parsers like SLR(1), LALR(1), and operator precedence, and automated top-down parsers like LL(1) parsers. If you need a detailed explanation of one of these techniques, we suggest that you consult the older textbooks listed in the chapter bibliography for an explanation of how those techniques differ from LR(1).

3.2 Expressing Syntax

A parser is, essentially, an engine for determining whether or not the input program is a valid sentence in the source language. To answer this question, we need both a formal mechanism for specifying the syntax of the input language, and a systematic method of determining membership in the formally-specified language. This section describes one mechanism for expressing syntax: a simple variation on the Backus-Naur form for writing formal grammars. The remainder of the chapter discusses techniques for determining membership in the language described by a formal grammar.

3.2.1 Context-Free Grammars

The traditional notation for expressing syntax is a *grammar*—a collection of rules that define, mathematically, when a string of symbols is actually a sentence in the language.

Computer scientists usually describe the syntactic structure of a language using an abstraction called a *context-free grammar* (CFG). A CFG, G, is a set of rules that describe how to form sentences; the collection of sentences that can be derived from G is called the *language defined by* G, and denoted $L(G)$. An example may help. Consider the following grammar, which we call SN:

$$\begin{array}{rcl} \textit{SheepNoise} & \rightarrow & \textit{SheepNoise}\ \underline{\texttt{baa}} \\ & | & \underline{\texttt{baa}} \end{array}$$

The first rule reads "*SheepNoise* can derive the string *SheepNoise* baa," where *SheepNoise* is a syntactic variable and baa is a word in the language described by the grammar. The second rule reads "*SheepNoise* can also derive the string baa."

To understand the relationship between the SN grammar and $L(SN)$, we need to specify how to apply the rules in the grammar to derive sentences in $L(SN)$. To begin, we must identIfy the *goal symbol* or *start symbol* of SN. The goal symbol represents the set of all strings in $L(SN)$. As such, it cannot be one of the words in the language. Instead, it must be one of the syntactic variables introduced to add structure and abstraction to the language. Since SN has only one syntactic variable, *SheepNoise* must be the goal symbol.

To derive a sentence, we begin with the string consisting of just the goal symbol. Next, we pick a syntactic variable, α, in the string and a rule $\alpha \rightarrow \beta$ that has α on its left-hand side. We rewrite the string by replacing the selected occurrence of α with the right-hand side of the rule, β. We repeat this process until the string contains no more syntactic variables; at this point, the string consists entirely of words in the language, or terminal symbols.

At each point in this derivation process, the string is a collection of symbols drawn from the union of the set of syntactic variables and the set of words in the language. A string of syntactic variables and words is considered a *sentential form* if some valid sentence can be derived from it—that is, if it occurs in some step of a valid derivation. If we begin with *SheepNoise* and apply successive rewrites using the two rules, at each step in the process the string will be a sentential form. When we have reached the point where the string contains only words in the language (and no syntactic variables), the string is a sentence in $L(SN)$.

For *SN*, we must begin with the string "*SheepNoise*." Using rule two, we can rewrite *SheepNoise* as `baa`. Since the sentential form contains only terminal symbols, no further rewrites are possible. Thus, the sentential form "`baa`" is a valid sentence in the language defined by our grammar. We can represent this derivation in tabular form.

Rule	*Sentential Form*
	SheepNoise
2	`baa`

We could also begin with *SheepNoise* and apply rule one to obtain the sentential form "*SheepNoise* `baa`". Next, we can use rule two to derive the sentence "`baa` `baa`".

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

As a notational convenience, we will build on this interpretation of the symbol →; when convenient, we will write $\rightarrow^+$ to mean "derives in one or more step." Thus, we might write *SheepNoise* $\rightarrow^+$ `baa` `baa`.

Of course, we can apply rule one in place of rule two to generate an even longer string of `baa`s. Repeated application of this pattern of rules, in a sequence (*rule one*)* *rule two* will derive the language consisting of one or more occurrences of the word `baa`. This corresponds to the set of noises that a sheep makes, under normal circumstances. These derivations all have the same form.

Rule	*Sentential Form*
	SheepNoise
1	*SheepNoise* `baa`
1	*SheepNoise* `baa` `baa`
	. . . and so on . . .
1	*SheepNoise* `baa` . . . `baa`
2	`baa` `baa` . . . `baa`

Notice that this language is equivalent to the RE `baa` `baa`* or `baa`$^+$.

More formally, a grammar G is a four-tuple, $G = (T, NT, S, P)$, where:

T is the set of *terminal symbols*, or words, in the language. Terminal symbols are the fundamental units of grammatical sentences. In a compiler, the terminal symbols correspond to words discovered in lexical analysis.

NT is the set of *non-terminal symbols*, or *syntactic variables*, that appear in the rules of the grammar. NT consists of all the symbols mentioned in the rules other than those in T. Non-terminal symbols are variables used to provide abstraction and structure in the set of rules.

S is a designated member of NT called the *goal symbol* or *start symbol*. Any derivation of a sentence in G must begin with S. Thus, the language derivable from G (denoted $L(G)$) consists of exactly the sentences that can be derived starting from S. In other words, S represents the set of valid sentences in $L(G)$.

P is a set of *productions* or *rewrite rules*. Formally, $P : NT \rightarrow (T \cup NT)^*$. Notice that we have restricted the definition so that it allows only a single non-terminal on the lefthand side. This ensures that the grammar is *context free*.

The rules of P encode the syntactic structure of the grammar.

Notice that we can derive NT, T, and P directly from the grammar rules. For the *SN* grammar, we can also discover S. In general, discovering the start symbol is harder. Consider, for example, the grammar:

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

The grammar describes the set of sentences consisting of balanced pairs of alternating parentheses and square brackets. It is not clear, however, if the outermost pair should be parentheses or square brackets. Designating *Paren* as S forces outermost parentheses. Designating *Bracket* as S forces outermost square brackets. If the intent is that either can serve as the outermost pair of symbols, we need two additional productions:

$$\begin{array}{lcl} \mathit{Start} & \rightarrow & \mathit{Paren} \\ & | & \mathit{Bracket} \end{array}$$

This grammar has a clear and unambiguous goal symbol, *Start*. Because *Start* does not appear in the right-hand side of any production, it must be the goal symbol. Some systems that manipulate grammars require that a grammar have a single *Start* symbol that appears in no production's right-hand side. They use this property to simplify the process of discovering S. As our example shows, we can always create a unique start symbol by adding one more non-terminal and a few simple productions

3.2.2 Constructing Sentences

To explore the power and complexity of context-free grammars, we need a more involved example than *SN*. Consider the following grammar:

$$\begin{array}{r|lcl} 1. & \mathit{Expr} & \rightarrow & \mathit{Expr}\ \mathit{Op}\ \texttt{Number} \\ 2. & & | & \texttt{Number} \\ 3. & \mathit{Op} & \rightarrow & + \\ 4. & & | & - \\ 5. & & | & \times \\ 6. & & | & \div \end{array}$$

Digression: Notation for Context-Free Grammars

The traditional notation used by computer scientists to represent a context-free grammar is called Backus-Naur form, or BNF. BNF denoted non-terminals by wrapping them in angle brackets, like ⟨`SheepNoise`⟩. Terminal symbols were underlined. The symbol `::=` meant "derives", and the symbol | meant also derives. In BNF, our example grammar *SN* would be written:

$$\begin{array}{rcl} \langle\texttt{SheepNoise}\rangle & \texttt{::=} & \langle\texttt{SheepNoise}\rangle\ \underline{\texttt{baa}} \\ & | & \underline{\texttt{baa}} \end{array}$$

BNF has its origins in the late 1950's and early 1960's. The syntactic conventions of angle brackets, underlining, `::=` and | arose in response to the limited typographic options available to people writing language descriptions. (For an extreme example, see David Gries' book *Compiler Construction for Digital Computers*, which was printed entirely using one of the print trains available on a standard lineprinter.) Throughout this book, we use a slightly updated form of BNF. Non-terminals are written with *slanted text*. Terminals are written in the `typewriter font` (and underlined when doing so adds clarity). "Derives" is written with a rightward-pointing arrow.

We have also forsaken the use of `*` to represent multiply and `/` to represent divide. We opt for the standard algebraic symbols × and ÷, except in actual program text. The meaning should be clear to the reader.

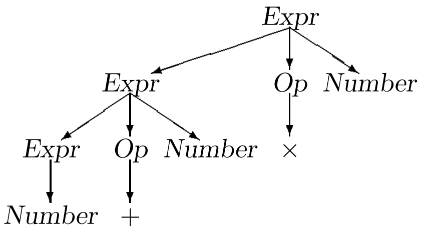
It defines a set of expressions over `Number`s and the four operators +, −, ×, and ÷. Using the grammar as a rewrite system, we can derive a large set of expressions. For example, applying rule 2 produces the trivial expression consisting solely of `Number`. Using the sequence 1, 3, 2 produces the expression `Number` + `Number`.

Rule	*Sentential Form*
	Expr
1	*Expr Op* `Number`
3	*Expr* + `Number`
2	`Number` + `Number`

Longer rewrite sequences produce more complex expressions. For example 1, 5, 1, 3, 2 derives the sentence `Number` + `Number` × `Number`.

Rule	*Sentential Form*
	Expr
1	*Expr Op* `Number`
5	*Expr* × `Number`
1	*Expr Op* `Number` × `Number`
3	*Expr* + `Number` × `Number`
2	`Number` + `Number` × `Number`

We can depict this derivation graphically.

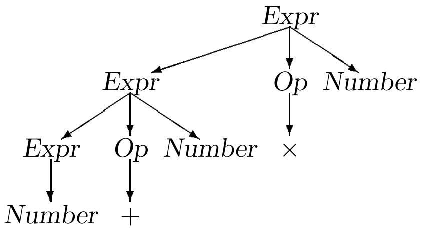


This derivation tree, or *syntax tree*, represents each step in the derivation.

So far, our derivations have always expanded the rightmost non-terminal symbol remaining in the string. Other choices are possible; the obvious alternative is to select the leftmost non-terminal for expansion at each point. Using leftmost choices would produce a different derivation sequence for the same sentence. For `Number` + `Number` × `Number`, the leftmost derivation would be:

Rule	*Sentential Form*
	Expr
1	*Expr Op* `Number`
1	*Expr Op* `Number` *Op* `Number`
2	`Number` *Op* `Number` *Op* `Number`
3	`Number` + `Number` *Op* `Number`
5	`Number` + `Number` × `Number`

This "leftmost" derivation uses the same set of rules as the "rightmost" derivation, but applies them in a different order. The corresponding derivation tree looks like:



It is identical to the derivation tree for the rightmost derivation! The tree represents all the rules applied in the derivation, but not their order of application.

We would expect the rightmost (or leftmost) derivation for a given sentence to be unique. If multiple rightmost (or leftmost) derivations exist for some sentence, then, at some point in the derivation, multiple distinct expansions of the rightmost (leftmost) non-terminal lead to the same sentence. This would produce multiple derivations and, possibly, multiple syntax trees—in other words, the sentence would lack a unique derivation.

A grammar G is *ambiguous* if and only if there exists a sentence in $L(G)$ that has multiple rightmost (or leftmost) derivations. In general, grammatical structure is related to the underlying meaning of the sentence. Ambiguity is often undesirable; if the compiler cannot be sure of the meaning of a sentence, it cannot translate it into a single definitive code sequence.

The classic example of an ambiguous construct in the grammar for a programming language arises in the definition of the `if-then-else` construct found in many Algol-like languages. The straightforward grammar for `if-then-else` might be:

$$\begin{array}{lcl}
\mathit{Stmt} & \rightarrow & \texttt{if (}\mathit{Expr}\texttt{) then }\mathit{Stmt}\texttt{ else }\mathit{Stmt} \\
 & | & \texttt{if (}\mathit{Expr}\texttt{) then }\mathit{Stmt} \\
 & | & \mathit{Assignment} \\
 & | & \ldots
\end{array}$$

This fragment shows that the `else` part is optional. Unfortunately, with this grammar the code fragment

$$\texttt{if (}\mathit{Expr}_1\texttt{) then if (}\mathit{Expr}_2\texttt{) then }\mathit{Stmt}_1\texttt{ else }\mathit{Stmt}_2\texttt{s}$$

has two distinct derivations. The difference between them is simple. Using indentation to convey the relationship between the various parts of the statements, we have:

$$\begin{array}{lll}
\texttt{if (}\mathit{Expr}_1\texttt{)} & \qquad\qquad & \texttt{if (}\mathit{Expr}_1\texttt{)} \\
\quad\texttt{then if (}\mathit{Expr}_2\texttt{)} & & \quad\texttt{then if (}\mathit{Expr}_2\texttt{)} \\
\qquad\texttt{then }\mathit{Stmt}_1 & & \qquad\texttt{then }\mathit{Stmt}_1 \\
\qquad\texttt{else }\mathit{Stmt}_2 & & \quad\texttt{else }\mathit{Stmt}_2
\end{array}$$

The version on the left has *Stmt*$_2$ controlled by the inner `if` statement, so it executes if *Expr*$_1$ is true and *Expr*$_2$ is false. The version on the right associates the `else` clause with the first `if` statement, so that *Stmt*$_2$ executes if *Expr*$_1$ is false (independently of *Expr*$_2$). Clearly, the difference in derivation will produce different behavior for the compiled code.

To remove this ambiguity, the grammar must be modified to encode a rule for determining which `if` controls an `else`. To fix the `if-then-else` grammar, we can rewrite it as:

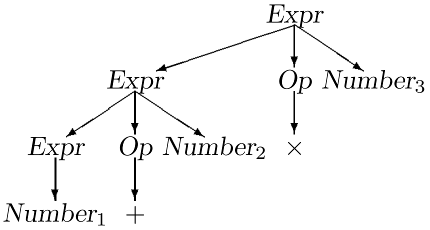
$$\begin{array}{lcl}
\mathit{Stmt} & \rightarrow & \mathit{WithElse} \\
 & | & \mathit{LastElse} \\
\mathit{WithElse} & \rightarrow & \texttt{if (}\mathit{Expr}\texttt{) then }\mathit{WithElse}\texttt{ else }\mathit{WithElse} \\
 & | & \mathit{Assignment} \\
 & | & \mathit{other\ statements}\ \ldots \\
\mathit{LastElse} & \rightarrow & \texttt{if (}\mathit{Expr}\texttt{) then }\mathit{Stmt} \\
 & | & \texttt{if (}\mathit{Expr}\texttt{) then }\mathit{WithElse}\texttt{ else }\mathit{LastElse}
\end{array}$$

The 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 unclosed `if`.

This ambiguity arises from a simple shortcoming of the grammar. The solution resolves the ambiguity in a way that is both easy to understand and easy for the programmer to remember. In Section 3.6.1, we will look at other kinds of ambiguity and systematic ways of handling them.

3.2.3 Encoding Meaning into Structure

The `if-then-else` ambiguity points out the relationship between grammatical structure and meaning. However, ambiguity is not the only situation where grammatical structure and meaning interact. Consider again the derivation tree for our simple expression, `Number` + `Number` × `Number`.



We have added subscripts to the instances of `Number` to disambiguate the discussion. A natural way to evaluate the expression is with a simple postorder treewalk. This would add $\mathtt{Number}_1$ and $\mathtt{Number}_2$ and use that result in the multiplication with $\mathtt{Number}_3$, producing $(\mathtt{Number}_1 + \mathtt{Number}_2) \times \mathtt{Number}_3$ This evaluation contradicts the rules of algebraic precedence taught in early algebra classes. Standard precedence would evaluate this expression as

$$\mathtt{Number}_1 + (\mathtt{Number}_2 \times \mathtt{Number}_3).$$

The expression grammar should have the property that it builds a tree whose "natural" treewalk evaluation produces this result.

The problem lies in the structure of the grammar. All the arithmetic operators derive in the same way, at the same level of the grammar. We need to restructure the grammar to embed the proper notion of precedence into its structure, in much the same way that we embedded the rule to disambiguate the `if-then-else` problem.

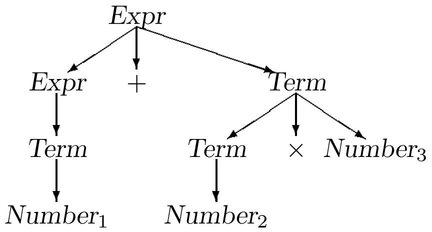
To introduce precedence into the grammar, we need to identify the appropriate levels of precedence in the language. For our simple expression grammar, we have two levels of precedence: lower precedence for + and −, and higher precedence for × and ÷. We associate a distinct non-terminal with each level of precedence and isolate the corresponding part of the grammar.

1.	*Expr*	→	*Expr* + *Term*
2.		|	*Expr* − *Term*
3.		|	*Term*
4.	*Term*	→	*Term* × `Number`
5.		|	*Term* ÷ `Number`
6.		|	`Number`

Here, *Expr* represents the lower level of precedence for + and −, while *Term* represents the higher level for × and ÷. Using this grammar to produce a rightmost derivation for the expression $\texttt{Number}_1 + \texttt{Number}_2 \times \texttt{Number}_3$, we get:

Rule	*Sentential Form*
	Expr
1	$\mathit{Expr} + \mathit{Term}$
4	$\mathit{Expr} + \mathit{Term} \times \texttt{Number}_3$
6	$\mathit{Expr} + \texttt{Number}_2 \times \texttt{Number}_3$
3	$\mathit{Term} + \texttt{Number}_2 \times \texttt{Number}_3$
6	$\texttt{Number}_1 + \texttt{Number}_2 \times \texttt{Number}_3$

This produces the following syntax tree:



A postorder treewalk over this syntax tree will first evaluate $\texttt{Number}_2 \times \texttt{Number}_3$ and then add $\texttt{Number}_1$ to the result. This corresponds to the accepted rules for precedence in ordinary arithmetic. Notice that the addition of non-terminals to enforce precedence adds interior nodes to the tree. Similarly, substituting the individual operators for occurrences of *Op* removes interior nodes from the tree.

To make this example grammar a little more realistic, we might add optional parentheses at the highest level of precedence. This requires introduction of another non-terminal and an appropriate rewrite of the grammar. If we also add support for both numbers and identifiers, the resulting grammar looks like:

1.	*Expr*	→	$\mathit{Expr} + \mathit{Term}$
2.		|	$\mathit{Expr} - \mathit{Term}$
3.		|	Term
4.	*Term*	→	$\mathit{Term} \times \mathit{Factor}$
5.		|	$\mathit{Term} \div \mathit{Factor}$
6.		|	Factor
7.	*Factor*	→	(*Expr*)
8.		|	`Number`
9.		|	`Identifier`

We will use this grammar as we explore parsing in the next several sections. We will refer to it as the *classic expression grammar*. In discussing automatic techniques, we may add one more production: *Goal*→*Expr*. Having a unique goal symbol simplifies some of the algorithms for automatically deriving parsers. For space reasons, we will often abbreviate `Number` as `Num` and `Identifier` as `Id`.

3.2.4 Discovering a Specific Derivation

We have seen how to discover sentences that are in $L(G)$ for our grammar G. By contrast, a compiler must infer a derivation for a given input string that, in fact, may not be a sentence. The process of constructing a derivation from a specific input sentence is called *parsing*. The compiler needs an automatic, algorithmic way to discover derivations. Since derivation trees are equivalent to derivations, we can think of parsing as building the derivation tree from the input string. Since parsing works by constructing a derivation tree, we often call that tree a *parse tree*.

The root of the derivation tree is fixed; its is a single node representing the goal symbol. The leaves of the tree are determined by the input string; the leaves must match the stream of classified words returned by the scanner. The remaining task is to discover an interior structure for the tree that connects the leaves to the root and is consistent with the rules embodied in the grammar. Two distinct and opposite approaches for constructing the tree suggest themselves.

Top-down parsers begin with the root and proceed by growing the tree toward the leaves. At each step, a top-down parser selects some non-terminal node and extends the tree downward from that node.

Bottom-up parsers begin with the leaves and proceed by growing the tree toward the root. At each step, a bottom-up parser adds nodes that extend the partially-built tree upward.

In either scenario, the parser makes a series of choices about which production to apply. Much of the intellectual complexity in parsing lies in the mechanisms for making these choices.

3.2.5 Context-Free Grammars versus Regular Expressions

To understand the differences between regular expressions and context-free grammars, consider the following two examples.

$$\begin{array}{c}
((\texttt{ident} \mid \texttt{num})\ op)^*\ (\texttt{ident} \mid \texttt{num}) \\
\texttt{op} \rightarrow + \mid - \mid \times \mid \div
\end{array}$$

$$\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
 & \mid & \texttt{Number} \\
 & \mid & \texttt{Identifier} \\
\mathit{Op} & \rightarrow & + \mid - \mid \times \mid \div
\end{array}$$

where `Identifier` and `Number` have their accustomed meanings. Both the RE and the CFG describe the same simple set of expressions.

To make the difference between regular expressions and context-free languages clear, consider the notion of a *regular grammar*. Regular grammars have the same expressive power as regular expressions—that is, they can describe the full set of regular languages.

A regular grammar is defined as a four-tuple, $R = (T, NT, S, P)$, with the same meanings as a context-free grammar. In a regular grammar, however, productions in P are restricted to one of two forms: $\alpha \rightarrow \mathsf{a}$, or $\alpha \rightarrow \mathsf{a}\beta$, where

$\alpha, \beta \in NT$ and $\mathtt{a} \in T$. In contrast, a context-free grammar allows productions with right-hand sides that contain an arbitrary set of symbols from $(T \cup NT)$. Thus, regular grammars are a prOper subset of context-free grammars. The same relationship holds for the regular languages and context-free languages.

(Expressing the difference as a RE, the regular grammar is limited to right-hand sides of the form $T \mid T\ NT$, while the context-free grammar allows right-hand sides of the form $(T \mid NT)^*$.)

Of course, we should ask: "are there interesting programming language constructs that can be expressed in a CFG but not a RG?" Many important features of modern programming languages fall into this gap between CFGs and RGs (or REs). Examples include matching brackets, like parentheses, braces, and pairs of keywords (*i.e.,* `begin` and `end`). Equally important, as the discussion in Section 3.2.3 shows, it can be important to shape the grammar so that it encodes specific information into the parse tree. For example, regular grammars cannot encode precedence, or the structure of an `if-then-else` construct. In contrast, all of these issues are easily encoded into a context-free grammar.

Since CFGs can recognize any construct specified by a RE, why use REs at all? The compiler writer could encode the lexical structure of the language directly into the CFG. The answer lies in the relative efficiency of DFA-based recognizers. Scanners based on DFA implementations take time proportional to the length of the input string to recognize words in the language. With reasonable implementation techniques, even the constants in the asymptotic complexity are small. In short, scanners are quite fast. In contrast, parsers for CFGs take time proportional to the length of the input, plus the length of the derivation. The constant overhead per terminal symbol is higher, as well.

Thus, compiler writers use DFA-based scanners for their efficiency, as well as their convenience. Moving micro-syntax into the context-free grammar would enlarge the grammar, lengthen the derivations, and make the front-end slower. In general, REs are used to classify words and to match patterns. When higher-level structure is needed, to match brackets, to impart structure, or to match across complex intervening context, CFGs are the tool of choice.

3.3 Top-Down Parsing

A top-down parser begins with the root of the parse tree and systematically extends the tree downward until it matches the leaves of the tree, which represent the classified words returned by the scanner. At each point, the process considers a partially-built parse tree. It selects a non-terminal symbol on the lower fringe of the tree and extends it by adding children that correspond to the right-hand side of some production for that non-terminal. It cannot extend the frontier from a terminal. This process continues until either the entire syntax tree is constructed, or a clear mismatch between the partial syntax tree and its leaves is detected. In the latter case, two possibilities exist. The parser may have selected the wrong production at some earlier step in the process; in which case backtracking will lead it to the correct choice. Alternatively, the input string may not be a valid sentence in the language being parsed; in this case,

```
token ← next_token
root ← start_symbol
node ← root

loop forever
    if node ∈ $T$ & node $\cong$ token then
        advance node to next node on the fringe
        token ← next_token
    else if node ∈ $T$ & node $\ncong$ token then
        backtrack
    else if node ∈ $NT$ then
        pick a rule "node→$\beta$"
        extend tree from node by building $\beta$
        node ← leftmost symbol in $\beta$

    if node is empty & token = EOF then
        accept
    else if node is empty & token ≠ EOF then
        backtrack
```

Figure 3.1: A leftmost, top-down parsing algorithm

backtracking will fail and the parser should report the syntactic error back to the user. Of course, the parser must be able to distinguish, eventually, between these two cases.

Figure 3.1 summarizes this process. The process works entirely on the lower fringe of the parse tree—which corresponds to the sentential forms we used in the examples in Section 3.2.2. We have chosen to expand, at each step, the leftmost non-terminal. This corresponds to a leftmost derivation. This ensures that the parser considers the words in the input sentence in the left-to-right order returned by the scanner.

3.3.1 Example

To understand the top-down parsing algorithm, consider a simple example: recognizing `x − 2 × y` as a sentence described by the classic expression grammar. The goal symbol of the grammar is *Expr*; thus the parser begins with a tree rooted in *Expr*. To show the parser's actions, we will expand our tabular representation of a derivation. The leftmost column shows the grammar rule used to reach each state; the center column shows the lower fringe of the partially constructed parse tree, which is the most recently derived sentential form. On the right, we have added a representation of the input stream. The ↑ shows the position of the scanner; it precedes the current input character. We have added two actions, → and ←, to represent advancing the input pointer and backtracking through the set of productions, respectively. The first several moves of the parser look like:

Rule or		
Action	*Sentential form*	*Input*
–	*Expr*	`↑x - 2 × y`
1	*Expr* + *Term*	`↑x - 2 × y`
3	*Term* + *Term*	`↑x - 2 × y`
6	*Factor* + *Term*	`↑x - 2 × y`
9	`Identifier` + *Term*	`↑x - 2 × y`
→	`Identifier` + *Term*	`x ↑- 2 × y`

The parser begins with *Expr*, the grammar's start symbol, and expands it, using rule 1. Since it is deriving a leftmost derivation, at each step, it considers the leftmost, unmatched symbol on the parse tree's lower fringe. Thus, it tries to rewrite *Expr* to derive `Identifier`. To do this, it rewrites the first non-terminal, *Expr*, into *Term* using rule 3. Then, it rewrites *Term* into *Factor* using rule 6, and *Factor* into `Identifier` using rule 9.

At this point, the leftmost symbol is a terminal symbol, so it checks for a match against the input stream. The words match, so it advances the input stream by calling the scanner (denoted by the → in the first column), and it moves rightward by one symbol along the parse tree's lower fringe.

After advancing, the parser again faces a terminal symbol as its leftmost unmatched symbol. It checks for a match against the current input symbol and discovers that the symbol + in the parse tree cannot match − in the input stream. At this point, one of two cases must hold:

1. The parser made a poor selection at some earlier expansion; if this is the case, it can backtrack and consider the alternatives for each choice already made.

2. The input string is not a valid sentence; if this is the case, the parser can only detect it by running out of possibilities in its backtracking.

In the example, the actual misstep occurred in the first expansion, when the parser rewrote *Expr* using rule 1. To correct that decision, the parser would need to retract the most recent rewrite, by rule 9, and try the other possibilities for expanding *Factor*. Of course, neither rule 7 nor rule 8 generate matches against the first input symbol, so it then backtracks on the expansion of *Term*, considering rules 4 and 5 as alternatives to rule 6. Those expansions will eventually fail, as well, since neither × nor ÷ match −. Finally, the parser will reconsider the rewrite of *Expr* with rule 1. When it tries rule 2, it can continue, at least for a while. In the derivation tale, we denote the entire backtracking sequence with the action "←". This line reads "backtrack to this sentential form and input state."

Rule or		
Action	*Sentential form*	*Input*
←	*Expr*	`↑x - 2 × y`
2	*Expr* − *Term*	`↑x - 2 × y`
3	*Term* − *Term*	`↑x - 2 × y`
6	*Factor* − *Term*	`↑x - 2 × y`
9	`Identifier` − *Term*	`↑x - 2 × y`
→	`Identifier` − *Term*	`x ↑- 2 × y`
→	`Identifier` − *Term*	`x - ↑2 × y`

Working from the expansion by rule 2, the parser has worked its way back to matching `Identifier` against `x` and advancing both the input symbol and the position on the fringe. Now, the terminal symbol on the fringe matches the input symbol, so it can advance both the fringe and the input symbol, again. At this point, the parser continues, trying to match *Term* against the current input symbol `2`.

Rule or		
Action	*Sentential form*	*Input*
6	`Identifier` − *Factor*	`x - ↑2 × y`
8	`Identifier` − `Number`	`x - ↑2 × y`
→	`Identifier` − `Number`	`x - 2 ↑× y`

The natural way to rewrite the fringe toward matching `2` is to rewrite by rule 6 and then rule 8. Now, the parser can match the non-terminal `Number` against the input symbol `2`. When it goes to advance the input symbol and the unmatched node on the fringe, it discovers that it has no symbols left on the fringe, but it has more input to consume. This triggers another round of backtracking, back to the rewrite of *Term*; when it rewrites *Term* with rule 4, it can proceed to a final and correct parse.

Rule or		
Action	*Sentential form*	*Input*
←	`Identifier` − *Term*	`x - ↑2 × y`
4	`Identifier` − *Term* × *Factor*	`x - ↑2 × y`
6	`Identifier` − *Factor* × *Factor*	`x - ↑2 × y`
8	`Identifier` − `Number` × *Factor*	`x - ↑2 × y`
→	`Identifier` − `Number` × *Factor*	`x - 2 ↑× y`
→	`Identifier` − `Number` × *Factor*	`x - 2 × ↑y`
8	`Identifier` − `Number` × `Number`	`x - 2 × ↑y`
→	`Identifier` − `Number` × `Number`	`x - 2 × y↑`

Finally, the parser has reached a configuration where it has systematically eliminated all non-terminals from the lower fringe of the parse tree, matched each leaf of the parse tree against the corresponding symbol in the input stream, and exhausted the input stream. This was the definition of success, so it has constructed a legal derivation for `x − 2 × y`.

3.3.2 Left Recursion

Clearly, the choice of a rewrite rule at each step has a strong impact on the amount of backtracking that the parser must perform. Consider another possible sequence of reductions for the same input string

Rule or		
Action	*Sentential form*	*Input*
–	*Expr*	↑x - 2 × y
1	*Expr* + *Term*	↑x - 2 × y
1	*Expr* + *Term* + *Term*	↑x - 2 × y
1	*Expr* + *Term* + ···	↑x - 2 × y
1	*Expr* + *Term* + ···	↑x - 2 × y
1	···	↑x - 2 × y

Here, the parser follows a simple rule. For each instance of a non-terminal, it cycles through the productions in the order that they appear in the grammar. Unfortunately, rule 1 generates a new instance of *Expr* to replace the old instance, so it generates an ever-expanding fringe without making any measurable progress.

The problem with this example arises from the combination of left recursion in the grammar and the top-down parsing technique. A production is said to be left-recursive if the first symbol on its right hand side is the same as its left-hand side, or if the left-hand side symbol appears in the right-hand side, and all the symbols that precede it can derive the empty string. Left recursion can produce non-termination in a top-down parser because it allows the algorithm to expand the parse tree's lower fringe indefinitely without generating a terminal symbol. Since backtracking is only triggered by a mismatch between a terminal symbol on the fringe and the current input symbol, the parser cannot recover from the expansion induced by left recursion.

We can mechanically transform a grammar to remove left recursion. For an obvious and immediate left recursion, shown on the left, we can rewrite it using right recursion as shown on the right.

$$\begin{array}{lcl} \textit{fee} & \rightarrow & \textit{fee}\ \alpha \\ & | & \beta \end{array} \qquad \begin{array}{lcl} \textit{fee} & \rightarrow & \beta\ \textit{fie} \\ \textit{fie} & \rightarrow & \alpha\ \textit{fie} \\ & | & \epsilon \end{array}$$

The transformation introduces a new non-terminal, *fie*, and transfers the recursion onto *fie*. It also adds the rule $\textit{fie} \rightarrow \epsilon$, where ϵ represents the empty string. This ϵ-production requires careful interpretation in the parsing algorithm. If the parser expands by the rule $\textit{fie} \rightarrow \epsilon$, the effect is to advance the current `node` along the tree's fringe by one position.

For a simple, immediate left recursion, we can directly apply the transformation. In our expression grammar, this situation arises twice—in the rules for *Expr* and the rules for *Term*;

Original			*Transformed*		
Expr	→	*Expr* + *Term*	*Expr*	→	*Term Expr′*
	$\mid$	*Expr* − *Term*	*Expr′*	→	+ *Term Expr′*
	$\mid$	*Term*		$\mid$	− *Term Expr′*
				$\mid$	ϵ
Term	→	*Term* × *Factor*	*Term*	→	*Factor Term′*
	$\mid$	*Term* ÷ *Factor*	*Term′*	→	× *Factor Term′*
	$\mid$	*Factor*		$\mid$	÷ *Factor Term′*
				$\mid$	ϵ

Plugging these replacements into the classic expression grammar yields:

1.	*Expr*	→	*Term Expr′*
2.	*Expr′*	→	+ *Term Expr′*
3.		$\mid$	− *Term Expr′*
4.		$\mid$	ϵ
5.	*Term*	→	*Factor Term′*
6.	*Term′*	→	× *Factor Term′*
7.		$\mid$	÷ *Factor Term′*
8.		$\mid$	ϵ
9.	*Factor*	→	(*Expr*)
10.		$\mid$	`Number`
11.		$\mid$	`Identifier`

This grammar describes the same set of expressions as the classic expression grammar. It uses right-recursion. It retains the left-associativity of the original grammar. It should work well with a top-down parser.

Rule or		
Action	*Sentential Form*	*Input*
–	*Expr*	↑ `x` − `2` × `y`
1	*Term Expr′*	↑ `x` − `2` × `y`
5	*Factor Term′ Expr′*	↑ `x` − `2` × `y`
11	`Id` *Term′ Expr′*	↑ `x` − `2` × `y`
→	`Id` *Term′ Expr′*	`x` ↑− `2` × `y`
8	`Id` *Expr′*	`x` ↑− `2` × `y`
3	`Id` − *Term Expr′*	`x` ↑− `2` × `y`
→	`Id` − *Term Expr′*	`x` − ↑`2` × `y`
5	`Id` − *Factor Term′ Expr′*	`x` − ↑`2` × `y`
10	`Id` − `Num` *Term′ Expr′*	`x` − ↑`2` × `y`
→	`Id` − `Num` *Term′ Expr′*	`x` − `2` ↑× `y`
6	`Id` − `Num` × *Factor Term′ Expr′*	`x` − `2` ↑× `y`
→	`Id` − `Num` × *Factor Term′ Expr′*	`x` − `2` × ↑ `y`
11	`Id` − `Num` × `Id` *Term′ Expr′*	`x` − `2` × ↑ `y`
→	`Id` − `Num` × `Id` *Term′ Expr′*	`x` − `2` × `y` ↑
8	`Id` − `Num` × `Id` *Expr′*	`x` − `2` × `y` ↑
4	`Id` − `Num` × `Id`	`x` − `2` × `y` ↑

```
arrange the non-terminals in some order
    $A_1, A_2, \ldots, A_n$

for i ← 1 to n
    for j ← 1 to i-1
        replace each production of the form
            $A_i \rightarrow A_j \gamma$ with the productions
            $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$,
            where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
            are all the current $A_j$ productions.
    eliminate any immediate left recursion on $A_i$
        using the direct transformation
```

Figure 3.2: Removal of left recursion

The transformation shown above eliminates immediate left recursion. Left recursion can occur indirectly, when a chain of rules such as $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$, and $\gamma \rightarrow \alpha\delta$ combines to create the situation that $\alpha \rightarrow^+ \alpha\delta$. This indirect left recursion is not always obvious; it can be hidden through an arbitrarily long chain of productions.

To convert these indirect left recursions into right recursion, we need a more systematic approach than inspection followed by application of our transformation. Figure 3.2 shows an algorithm that achieves this goal. It assumes that the grammar has no cycles ($A \rightarrow^+ A$) or ϵ productions ($A \rightarrow \epsilon$).

The algorithm works by imposing an arbitrary order on the non-terminals. The outer loop cycles through the non-terminals in this order, while the inner loop ensures that a production expanding A_i has no non-terminal A_j with $j < i$. When it encounters such a non-terminal, it forward substitutes the non-terminal away. This eventually converts each indirect left recursion into a direct left recursion. The final step in the outer loop converts any direct recursion on A_i to right recursion using the simple transformation shown earlier. Because new non-terminals are added at the end of the order 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 more clear. At the start of the i^{th} outer loop iteration

$$\forall\ k < i,\ \nexists \text{ a production expanding } A_k \text{ with } A_l \text{ in its } rhs, \text{ for } l < k.$$

At the end of this process, ($i = n$), all indirect left recursion has been eliminated through the repetitive application of the inner loop, and all immediate left recursion has been eliminated in the final step of each iteration.

3.3.3 Predictive Parsing

When we parsed `x - 2 × y` with the right-recursive expression grammar, we did not need to backtrack. In fact, we can devise a parser for the right-recursive expression grammar that never needs to backtrack. To see this, consider how the parser makes a decision that it must retract through backtracking.

The critical choice occurs when the parser selects a production with which to expand the lower fringe of the partially constructed parse tree. When it tries to expand some non-terminal α, it must pick a rule $\alpha \rightarrow \beta$. The algorithm, as shown in Figure 3.1, picks that rule arbitrarily. If, however, the parser could always pick the appropriate rule, it could avoid backtracking.

In the right-recursive variant of the expression grammar, the parser can make the correct choice by comparing the next word in the input stream against the right-hand sides. Look at the situation that arose in the derivation of `x` $-$ `2` $\times$ `y` in the previous section. When the parser state was in the state

Rule or Action	*Sentential Form*	*Input*
8	`Id` *Expr*$'$	`x` $\uparrow -$ `2` $\times$ `y`

it needed to choose an expansion for *Expr*$'$. The possible right-hand sides were: $+$ *Term Expr*$'$, $-$ *Term Expr*$'$, and ϵ. Since the next word in the input stream was $-$, the second choice is the only one that can succeed. The first choice generates a leading $+$, so it can never match and will lead directly to backtracking. Choosing ϵ can only match the end of string, since *Expr*$'$ can only occur at the right end of a sentential form.

Fortuitously, this grammar has a form where the parser can predict the correct expansion by comparing the possible right-hand sides against the next word in the input stream. We say that such a grammar is *predictive*; parsers built on this property are called predictive parsers.

Before going further, we should define the property that makes a grammar predictively parsable. For any two productions, $A \rightarrow \alpha \mid \beta$, the set of initial terminal symbols derivable from α must be distinct from those derivable from β. If we define $\textsc{First}(\alpha)$ as the set of tokens that can appear as the first symbol in some string derived from α, then we want

$$\textsc{First}(\alpha) \;\cap\; \textsc{First}(\beta) \;=\; \emptyset$$

For an entire grammar G, the desired property is

$$\begin{array}{l} \forall \text{ rules } A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \cdots \alpha_n \text{ in a grammar } G \\ \qquad \textsc{First}(\alpha_1) \cap \textsc{First}(\alpha_2) \cap \textsc{First}(\alpha_3) \cap \cdots \textsc{First}(\alpha_n) \;=\; \emptyset \end{array}$$

If this property holds true for each non-terminal in the grammar, then the grammar can be parsed predictively. Unfortunately, not all grammars are predictive.[1]

Consider, for example, a slightly more realistic version of our on-going example. A natural extension to our right recursive expression grammar would replace the production *Factor*$\rightarrow$`Identifier` with a set of productions that describe the syntax for scalar variable references, array variable references, and function calls.

[1] This condition is also called the $LL(1)$ condition. Any grammar that meets this condition can be used to construct a table-driven, $LL(1)$ parser.

```
For each $NT$ $A \in G$
    find the longest prefix $\alpha$ common to
        two or more right-hand sides
    if $\alpha \neq \epsilon$ then
        replace the rules expanding $A$,
            $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$
        with
            $A \rightarrow \alpha$ fie $\mid \gamma$
            fie $\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$
        and add fie to $NT$
Repeat until no common prefixes remain.
```

Figure 3.3: Left Factoring a Grammar

11.	*Factor*	→	`Identifier`
12.		|	`Identifier` `[` *Exprlist* `]`
13.		|	`Identifier` `(` *Exprlist* `)`
20.	*Exprlist*	→	*Expr* `,` *Exprlist*
21.		|	*Expr*

Here, we show the C syntactic convention of using parentheses for function calls and square brackets for array references. This grammar fragment is not predictive, because the initial terminal symbol generated in rules 11, 12, and 13 is the same. Thus, the parser, in trying to expand a *Factor* on the parse tree's lower fringe, cannot decide between 11, 12, and 13 on the basis of a single word lookahead. Of course, looking ahead two tokens would allow it to predict the correct expansion.

We can rewrite rules 11, 12, and 13 to make them predictive.

11.	*Factor*	→	`Identifier` *arguments*
12.	*arguments*	→	`[` *Exprlist* `]`
13.		|	`(` *Exprlist* `)`
14.		|	ϵ

In this case, we were able to transform the grammar into a predictive grammar. In essence, we introduced a new non-terminal to represent the common prefix of the three rules that were not predictive. We can apply this transformation, systematically and mechanically, to an arbitrary grammar. Figure 3.3 shows a simple algorithm that does this. However, left factoring the grammar will not always produce a predictive grammar.

Not all languages can be expressed in a predictive grammar. Using left-recursion elimination and left factoring, we may be able to transform a grammar to the point where it can be predictively parsed. In general, however, it is undecidable whether or not a predictive grammar exists for an arbitrary context-free language. For example, the language

$$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$$

has no predictive grammar.

3.3.4 Top-Down Recursive Descent Parsers

Given a predictive grammar G, we can construct a hand-coded parser for G that operates by recursive descent. A recursive descent parser is structured as a set of mutually recursive routines, one for each non-terminal in the grammar. The routine for non-terminal A recognizes an instance of A in the input stream. To accomplish this, the routine invokes other routines to recognize the various non-terminals on A's right-hand side.

Consider a set of productions $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$. Since G is predictive, the parser can select the appropriate right hand side (one of β_1, β_2, or β_3) using only the input token and the FIRST sets. Thus, the code in the routine for A should have the form:

```
/* find an $A$ */
if (current_token $\in$ FIRST($\beta_1$))
    find a $\beta_1$ & return true
else if (current_token $\in$ FIRST($\beta_2$))
    find a $\beta_2$ & return true
else if (current_token $\in$ FIRST($\beta_3$))
    find a $\beta_3$ & return true
else {
    report an error based on $A$ and current_token
    return false
}
```

For each right hand side $A \rightarrow \beta_i$, the routine needs to recognize each term in β_i. The code must check for each term, in order.

- For a terminal symbol, the code compares the input symbol against the specified terminal. If they match, it advances the input token and checks the next symbol on the right hand side. If a mismatch occurs, the parser should report the syntax error in a suitably informative message.

- For a non-terminal symbol, the code invokes the routine that recognizes that non-terminal. That routine either returns true as an indication of success, or it reports an error to the user and returns false. Success allows the parser to continue, recognizing the next symbol on the right hand side, if any more exist. A return value of false forces the routine to return false to its calling context.

For a right hand side $\beta_1 = \gamma\delta\rho$, with $\gamma, \rho \in NT$ and $\delta \in T$, the code needs to recognize a γ, a δ, and an ρ (abstracted away as "find a β_1 and return true" in the previous code fragment). This code might look like:

```
if (current_token ∈ FIRST($\beta_1$)) {
    if (Parse_$\gamma$() = false)
        return false
    else if (current_token ≠ $\delta$) {
        report an error finding $\delta$ in $A \rightarrow \gamma\delta\rho$
        return false
    }
    current_token ← next_token()
    if (Parse_$\rho$() = false)
        return false
    else
        return true
}
```

The routine Parse_A will contain a code fragment like this for each alternative right-hand side for A.

To construct a complete recursive descent parser, then, the strategy is clear. For each non-terminal, we construct a routine that recognizes that non-terminal. Each routine relies on the other routines to recognize non-terminals, and directly tests the terminal symbols that arise in its own right hand sides. Figure 3.4 shows a top-down recursive descent parser for the predictive grammar that we derived in Section 3.3.2. Notice that it repeats code for similar right hand sides. For example, the code in ExprP() under the tests for '+' and for '-' could be combined to produce a smaller parser.

Automating the Process Top-down recursive-descent parsing is usually considered a technique for hand coding a parser. Of course, we could build a parser generator that automatically emitted top-down recursive descent parsers for suitable grammars. The parser generator would first construct the necessary FIRST sets for each grammar symbol, check each non-terminal to ensure that the FIRST sets of its alternative right-hand sides are disjoint, and emit a suitable parsing routine for each non-terminal symbol in the grammar. The resulting parser would have the advantages of top-down recursive-descent parsers, such as speed, code-space locality, and good error detection. It would also have the advantages of a grammar-generated system, such as a concise, high-level specification and a reduced implementation effort.

3.4 Bottom-up Parsing

To parse a stream of words bottom-up, the parser begins by creating leaves in the parse tree for each word. This creates the base of the parse tree. To construct a derivation, it adds layers of non-terminals on top of the leaves, in a structure dictated by both the grammar and the input stream. The upper edge of this partially-constructed parse tree is called its *upper frontier*. Each layer extends the frontier upward, toward the tree's root.

To do this, the parser repeatedly looks for a part of that upper frontier that matches the right-hand side of a production in the grammar. When it finds a

```
Main()
  token $\leftarrow$ next_token();
  if (Expr() $\neq$ false)
      then
          next compilation step
  else return false

Expr()
  result $\leftarrow$ true
  if (Term() = false)
      then result $\leftarrow$ false
  else if (EPrime() = false)
      then result $\leftarrow$ false

  return result

EPrime()
  result $\leftarrow$ true
  if (token = '+') then
      token $\leftarrow$ next_token()
      if (Term() = false)
          then result $\leftarrow$ false
      else if (EPrime() = false)
          then result $\leftarrow$ false
  else if (token = '-') then
      token $\leftarrow$ next_token()
      if (Term() = false)
          then result $\leftarrow$ false
      else if (EPrime() = false)
          then result $\leftarrow$ false
  else result $\leftarrow$ true /* $\epsilon$ */

  return result

Term()
  result $\leftarrow$ true
  if (Factor = false)
      then result $\leftarrow$ false
  else if (TPrime() = false)
      then result $\leftarrow$ false

  return result
```

```
TPrime()
  result $\leftarrow$ true
  if (token = $\times$) then
      token $\leftarrow$ next_token()
      if (Factor() = false)
          then result $\leftarrow$ false
      else if (TPrime() = false)
          then result $\leftarrow$ false
  else if (token = $\div$) then
      token $\leftarrow$ next_token()
      if (Factor() = false)
          then result $\leftarrow$ false
      else if (TPrime() = false)
          then result $\leftarrow$ false;
  else result $\leftarrow$ true /* $\epsilon$ */

  return result


Factor()
  result $\leftarrow$ true
  if (token = '(') then
      token $\leftarrow$ next_token()
      if (Expr() = false)
          then result $\leftarrow$ false
      else if (token $\neq$ ')')
          then
              report syntax error
              result $\leftarrow$ false
          else token $\leftarrow$ next_token()
  else if (token = Number)
      then token $\leftarrow$ next_token()
  else if (token = identifier)
      then token $\leftarrow$ next_token()
  else
      report syntax error
      result $\leftarrow$ false

  return result
```

Figure 3.4: Recursive descent parser for expressions

Digression: Predictive parsers versus DFA*s*

Predictive parsing is the natural extension of DFA-style reasoning to parsers. A DFA makes its transition based on the next input character. A predictive parser requires that the expansions be uniquely determined by the next word in the input stream. Thus, at each non-terminal in the grammar, there must be a unique mapping from the first token in any acceptable input string to a specific production that leads to a derivation for that string. The real difference in power between a DFA and a predictively-parsable, or LL(1), grammar, derives from the fact that one prediction may lead to a right-hand side with many symbols, whereas, in a regular grammar, it predicts only a single symbol. This lets predictive grammars include productions like

$$p \rightarrow \underline{(}\ p\ \underline{)}$$

which is beyond the power of a regular expression to describe. (Recall that a regular expression can recognize $\underline{(}^+ \ \Sigma^* \ \underline{)}^+$, but this does not specify that the opening and closing parentheses must match.)

Of course, a hand-constructed top-down, recursive-descent parser can use arbitrary tricks to disambiguate production choices. For example, if a particular left-hand side cannot be predicted with a single symbol lookahead, the parser could use two symbols. Done judiciously, this should not cause problems.

match, it builds a node to represent the non-terminal symbol on production's left-hand side, and adds edges to the nodes representing elements of the right-hand side. This extends the upper frontier. This process terminates under one of two conditions.

1. The parser reduces the upper frontier to a single node that represents the grammar's start symbol. If all the input has been consumed, the input stream is a valid sentence in the language.

2. No match can be found. Since, the parser has been unable to build a derivation for the input stream, the input is not a valid sentence. The parser should issue an appropriate error message.

A successful parse runs through every step of the derivation. A failed parse halts when it can find no further steps, at which point it can use the context accumulated in the tree to produce a meaningful error message.

Derivations begin with the goal symbol and work towards a sentence. Because a bottom-up parser proceeds bottom-up in the parse tree, it discovers derivation steps in reverse order. Consider a production $\alpha \rightarrow \beta$ where $\beta \ \in \ T$. A bottom-up parser will "discover" the derivation step $\alpha \rightarrow \beta$ before it discovers the step that derives α. Thus, if a derivation consists of a series of steps that produces a series of sentential forms

$$S_0 = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \textit{sentence},$$

the bottom-up parser will discover $\gamma_{n-1} \Rightarrow \gamma_n$ before it discovers $\gamma_{n-2} \Rightarrow \gamma_{n-1}$. (It must add the nodes implied by $\gamma_{n-1} \Rightarrow \gamma_n$ to the frontier before it can discover any matches that involve those nodes. Thus, it cannot discover the nodes in an order inconsistent with the reverse derivation.) At each point, the parser will operate on the frontier of the partially constructed parse tree; the current frontier is a prefix of the corresponding sentential form in the derivation. Because the sentential form occurs in a rightmost derivation, the missing suffix consists entirely of terminal symbols.

Because the scanner considers the words in the input stream in a left-to-right order, the parser should look at the leaves from left to right. This suggests a derivation order that produces terminals from right to left, so that its reverse order matches the scanner's behavior. This leads, rather naturally, to bottom-up parsers that construct, in reverse, a rightmost derivation.

In this section, we consider a specific class of bottom-up parsers called LR(1) parsers. LR(1) parsers scan the input from left-to-right, the order in which scanners return classified words. LR(1) parsers build a rightmost derivation, *in reverse*. LR(1) parsers make decisions, at each step in the parse, based on the history of parse so far, and, at most, a *lookahead* of one symbol. The name LR(1) derives from these three properties: left-to-right scan, reverse-rightmost derivation, and 1 symbol of lookahead.[2] 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 only one symbol of lookahead to determine parsing actions.

3.4.1 Using handles

The key to bottom-up parsing lies in using an efficient mechanism to discover matches along the tree's current upper frontier. Formally, the parser must find some substring $\beta\gamma\delta$ of the upper frontier where

1. $\beta\gamma\delta$ is the right-hand side of some production $\alpha \rightarrow \beta\gamma\delta$, and
2. $\alpha \rightarrow \beta\gamma\delta$ is one step in the rightmost derivation of the input stream.

It must accomplish this while looking no more than one word beyond the right end of $\beta\gamma\delta$.

We can represent each potential match as a pair $\langle \alpha \rightarrow \beta\gamma\delta, k \rangle$, where $\alpha \rightarrow \beta\gamma\delta$ is a production in G and k is the position on the tree's current frontier of the right end of δ. If replacing the occurrence of $\beta\gamma\delta$ that ends at k with α is the next step in the reverse rightmost derivation of the input string, then $\langle \alpha \rightarrow \beta\gamma\delta, k \rangle$ is a *handle* of the bottom-up parse. A handle concisely specifies the next step in building the reverse rightmost derivation.

A bottom-up parser operates by repeatedly locating a handle on the frontier of the current partial parse tree, and performing the reduction that it specifies.

[2] The theory of LR parsing defines a family of parsing techniques, the LR(k) parsers, for arbitrary $k \geq 0$. Here, k denotes the amount of lookahead that the parser needs for decision making. LR(1) parsers accept the same set of languages as LR(k) parsers.

	Token	Frontier	Handle	Action
1.	`Id`		— *none* —	*extend*
2.	−	`Id`	⟨*Factor*→`Id`,1⟩	*reduce*
3.	−	*Factor*	⟨*Term*→*Factor*,1⟩	*reduce*
4.	−	*Term*	⟨*Expr*→*Term*⟩	*reduce*
5.	−	*Expr*	— *none* —	*extend*
6.	`Num`	*Expr* −	— *none* —	*extend*
7.	×	*Expr* − `Num`	⟨*Factor*→`Num`,3⟩	*reduce*
8.	×	*Expr* − *Factor*	⟨*Term*→*Factor*⟩	*reduce*
9.	×	*Expr* − *Term*	— *none* —	*extend*
10.	`Id`	*Expr* − *Term* ×	— *none* —	*extend*
11.	EOF	*Expr* − *Term* × `Id`	⟨*Factor*→`Id`⟩	*reduce*
12.	EOF	*Expr* − *Term* × *Factor*	⟨*Term*→*Term* × *Factor*⟩	*reduce*
13.	EOF	*Expr* − *Term*	⟨*Expr*→*Expr* − *Term*⟩	*reduce*
14.	EOF	*Expr*	⟨*Goal*→*Expr*⟩	*reduce*
15.	EOF	*Goal*	— *none* —	*accept*

Figure 3.5: States of the bottom-up parser on `x − 2 × y`

When the frontier does not contain a handle, the parser extends the frontier by adding a single non-terminal to the right end of the frontier. To see how this works, consider parsing the string `x − 2 × y` using the classic expression grammar. At each step, the parser either finds a handle on the frontier, or it adds to the frontier. The state of the parser, at each step, is summarized in Figure 3.5, while Figure 3.6 shows the corresponding partial parse tree for each step in the process; the trees are drawn with their frontier elements justified along the top of each drawing.

As the example shows, the parser only needs to examine the upper frontier of partially constructed parse tree. Using this fact, we can build a particularly clean form of bottom-up parser, called a *shift-reduce* parser. These parsers use a stack to hold the frontier; this simplifies the algorithm in two ways. First, the stack trivializes the problem of managing space for the frontier. To extend the frontier, the parser simply shifts the current input symbol onto the stack. Second, it ensures that all handles occur with their right end at the top of the stack; this eliminates any overhead from tracking handle positions.

Figure 3.7 shows a simple shift-reduce parser. To begin, it shifts an invalid symbol onto the stack and gets the first input symbol from the scanner. Then, it follows the handle-pruning discipline: it shifts symbols from the input onto the stack until it discovers a handle, and it reduces handles as soon as they are found. It halts when it has reduced the stack to *Goal* on top of INVALID and consumed all the input.

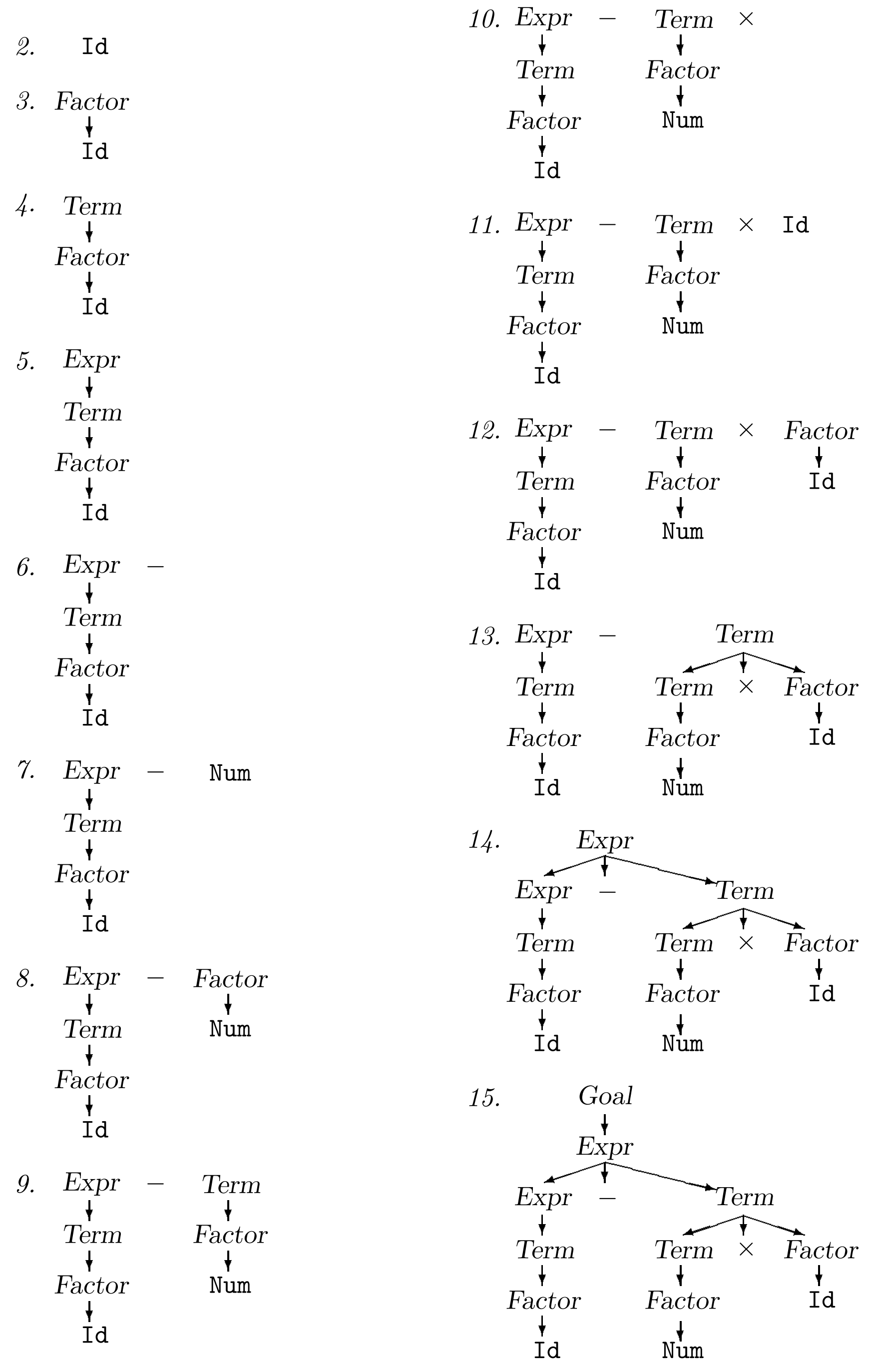

2. Id
3. Factor Id
4. Term Factor Id
5. Expr Term Factor Id
6. Expr – Term Factor Id
7. Expr – Num Term Factor Id
8. Expr – Factor Term Num Factor Id
9. Expr – Term Term Factor Factor Num Id
10. Expr – Term × Term Factor Factor Num Id
11. Expr – Term × Id Term Factor Factor Num Id
12. Expr – Term × Factor Term Factor Id Factor Num Id
13. Expr – Term Term Term × Factor Factor Factor Id Id Num
14. Expr Expr – Term Term Term × Factor Factor Factor Id Id Num
15. Goal Expr Expr – Term Term Term × Factor Factor Factor Id Id Num

Figure 3.6: Bottom-up parse of x – 2 × y

```
push INVALID
token ← next_token()
repeat until (top of stack = $Goal$ & token = EOF)
    if we have a handle $\alpha \rightarrow \beta$ on top of the stack
        then reduce by $\alpha \rightarrow \beta$
            pop $|\ \beta\ |$ symbols off the stack
            push $\alpha$ onto the stack
        else if (token $\neq$ EOF)
            then shift
                push token
                token ← next_token()
        else
            report syntax error & halt
```

Figure 3.7: Shift-reduce parsing algorithm

Using this algorithm, Figure 3.5 can be reinterpreted to show the actions of our shift-reduce parser on the input stream `x − 2 × y`. The row labelled Token shows the contents of the variable `token` in the algorithm. The row labelled Frontier depicts the contents of the stack at each step; the stack top is to the right. Finally, the action *extend* indicates a shift; *reduce* still indicates a reduction.

For an input stream of length s, this parser must perform s shifts. It must perform a reduction for each step in the derivation, for r steps. It looks for a handle on each iteration of the while loop, so it must perform $s+r$ handle-finding operations. If we can keep the cost of handle-finding to a small constant, we have a parser that can operate in time proportional to the length of the input stream (in words) plus the length of the derivation. Of course, this rules out traversing the entire stack on each handle-find, so it places a premium on efficient handle-finding.

3.4.2 Finding Handles

The handle-finding mechanism is the key to efficient bottom-up parsing. Let's examine this problem in more detail. In the previous section, handles appeared in the example as if they were derived from an oracle. Lacking an oracle, we need an algorithm.

As it parses an input string, the parser must track multiple potential handles. For example, on every legal input, the parser eventually reduces to its goal symbol. In the classic expression grammar, this implies that the parser reaches a state where its has the handle $\langle Goal \rightarrow Expr, 1\rangle$ on its stack. This particular handle represents one half of the halting condition—having *Goal* as the root of the parse tree. (The only production reducing to *Goal* is $Goal \rightarrow Expr$. Since it must be the last reduction, the position must be 1.) Thus, $\langle Goal \rightarrow Expr\rangle$ is a potential handle at every step in the parse, from first to last.

In between, the parser discovers other handles. In the example of Figure 3.5, it found eight other handles. At each step, the set of potential handles represent all the legal completions of the sentential form that has already been recognized. (The sentential form consists of the upper frontier, concatenated with the remainder of the input stream—beginning with the current lookahead token.) Each of these potential handles contains the symbol on top of the stack; that symbol can be at a different location in each handle.

To represent the position of the top of stack in a potential handle, we introduce a placeholder, $\bullet$, into the right-hand side of the production. Inserting the placeholder into $\alpha \rightarrow \beta\gamma\delta$ gives rise to four different strings:

$$\alpha \rightarrow \bullet\beta\gamma\delta, \quad \alpha \rightarrow \beta \bullet \gamma\delta, \quad \alpha \rightarrow \beta\gamma \bullet \delta, \text{ and } \quad \alpha \rightarrow \beta\gamma\delta\bullet.$$

This notation captures the different relationships between potential handles and the state of the parser. In step 7 of the example, the parser has the frontier *Expr* $-$ `Num`, with the handle $\langle$*Factor*$\rightarrow$`Num`,3$\rangle$. Using $\bullet$, we can write this as $\langle$*Factor*$\rightarrow$`Num` $\bullet\rangle$; the position is implicit relative to the top of the stack. A potential handle becomes a handle when $\bullet$ appears at the right end of the production. It must also track a number of other potential handles.

For example, $\langle$*Expr*$\rightarrow$*Expr* $- \bullet$ *Term*$\rangle$ represents the possibility that `Num` will eventually reduce to *Term*, with a right context that allows the parser to reduce *Expr* $-$ *Term* to *Expr*. Looking ahead in the parse, this potential handle becomes the active handle in step 13, after `Num` $\times$ `Id` has been reduced to *Term*. Other potential handles at step 7 include $\langle$*Term*$\rightarrow \bullet$ *Factor*$\rangle$ and $\langle$*Goal*$\rightarrow \bullet$ *Expr*$\rangle$.

This notation does not completely capture the state of the parser. Consider the parser's action at step 9. The frontier contains *Expr* $-$ *Term*, with potential handles including $\langle$*Expr*$\rightarrow$*Expr* $-$ *Term* $\bullet\rangle$ and $\langle$*Term*$\rightarrow$*Term* $\bullet \times$ *Factor*$\rangle$. Rather than reduce by $\langle$*Expr*$\rightarrow$*Expr* $-$ *Term* $\bullet\rangle$, the parser extended the frontier to follow the future represented by $\langle$*Term*$\rightarrow$*Term* $\bullet \times$ *Factor*$\rangle$. The example demonstrates that this is the right action. Reducing would move the parser into a state where it could make no further progress, since it cannot reduce subsequently *Expr* $\times$ *Factor*.

To choose between these two actions, the parser needs more context than it has on the frontier. In particular, the parser can look ahead one symbol to discover that the next word in the input stream is $\times$. This single-word lookahead allows it to determine that the correct choice is pursuing the handle represented by $\langle$*Term*$\rightarrow$*Term* $\bullet \times$ *Factor*$\rangle$, rather than reducing as represented by $\langle$*Expr*$\rightarrow$*Expr* $-$ *Term* $\bullet\rangle$. The key to making this decision is the value of the next token, also called the lookahead symbol.

This suggests that we can represent each possible future decision of the parser as a pair, $\langle alpha \rightarrow \beta \bullet \gamma\delta, \texttt{a}\rangle$, where $\alpha \rightarrow \beta\gamma\delta \in P$, and $\texttt{a} \in T$. This pair, called an *item*, is interpreted as

> The parser is in a state where finding an α, followed by the terminal `a`, would be consistent with its left context. It has already found a

β, so finding $\gamma\delta\mathtt{a}$ would allow it to reduce by $\alpha\rightarrow\beta\gamma\delta$

At each step in the parse, a collection of these pairs represents the set of possible futures for the derivation, or the set of suffixes that would legally complete the left context that the parser has already seen. This pair is usually called an LR(1) item. When written as an LR(1) item, it has square brackets rather than angle brackets.

The set of LR(1) items is finite. If r is the maximal length of a right-hand side for any production in P, then the number of LR(1) items can be no greater than $(r+1)\cdot \mid T \mid$. For a grammar G, the set of LR(1) items includes all the possible handles for G. Thus, the set of handles is finite and can be recognized by a DFA. This is the central insight behind LR(1) parsers:

> *Because the set of* LR(1) *items is finite, we can build a handle-finder that operates as a* DFA.

The LR(1) parser construction algorithm builds the DFA to recognize handles. It uses a shift-reduce parsing framework to guide the application of the DFA. The framework can invoke the DFA recursively; to accomplish this, it stores information about internal states of the DFA on the stack.

3.5 Building an LR(1) parser

The LR(1) parsers, and their cousins SLR(1) and LALR(1), are the most widely used family of parsers. The parsers are easy to build and efficient. Tools to automate construction of the tables are widely available. Most programming language constructs have a natural expression in a grammar that is parsable with an LR(1) parser. This section explains how LR(1) parsers work, and shows how to construct the parse tables for one kind of LR(1) parser, a canonical LR(1) parser.

LR(1) table construction is an elegant application of theory to practice. However, the actual process of building tables is better left to parser generators than to humans. That notwithstanding, the algorithm is worth studying because it explains the kinds of errors that the parser generator can encounter, how they arise, and how they can be remedied.

3.5.1 The LR(1) parsing algorithm

An LR(1) parser consists of a skeleton parser and a pair of tables that drive the parser. Figure 3.8 shows the skeleton parser; it is independent of the grammar being parsed. The bottom half of the figure shows the ACTION and GOTO tables for the classic expression grammar without the production *Factor* → `(` *Expr* `)`.

The skeleton parser resembles the shift-reduce parser shown in Figure 3.7. At each step, it pushes two objects onto the stack: a grammar symbol from the frontier and a state from the handle recognizer. It has four actions:

1. shift: extends the frontier by shifting the lookahead token onto the stack, along with a new state for the handle recognizer. This may result in a recursive invocation of the recognizer.

```
push INVALID
push $s_0$
token $\leftarrow$ next_token()
while (true)
        s $\leftarrow$ top of stack
        if action[s,token] = "$shift\ s_i$"' then
              push token
              push $s_i$
              token $\leftarrow$ next_token()
        else if action[s,token] = "$reduce\ A \rightarrow \beta$" then
              pop $2 \times \mid \beta \mid$ symbols
              s $\leftarrow$ top of stack
              push $A$
              push goto[s,$A$]
        else if action[s, token] = "$accept$" then
              return
        else error()
```

The Skeleton LR(1) Parser

Action Table							
State	`EOF`	`+`	`-`	×	÷	`Num`	`id`
0						s 4	s 5
1	acc	s 6	s 7				
2	r 4	r 4	r 4	s 8	s 9		
3	r 7	r 7	r 7	r 7	r 7		
4	r 8	r 8	r 8	r 8	r 8		
5	r 9	r 9	r 9	r 9	r 9		
6						s 4	s 5
7						s 4	s 5
8						s 4	s 5
9						s 4	s 5
10	r 2	r 2	r 2	s 8	s 9		
11	r 3	r 3	r 3	s 8	s 9		
12	r 5	r 5	r 5	r 5	r 5		
13	r 6	r 6	r 6	r 6	r 6		

Goto Table		
Expr	*Term*	*Factor*
1	2	3
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	10	3
0	11	3
0	0	12
0	0	13
0	0	0
0	0	0
0	0	0
0	0	0

LR(1) Tables for the Classic Expression Grammar

Figure 3.8: An LR(1) parser

	Token	Stack	Action
1.	`Id`	`$ 0`	*shift*
2.	-	`$ 0 id 5`	*reduce*
3.	-	`$ 0 Factor 3`	*reduce*
4.	-	`$ 0 Term 2`	*reduce*
5.	-	`$ 0 Expr 1`	*shift*
6.	`Num`	`$ 0 Expr 1 - 7`	*shift*
7.	×	`$ 0 Expr 1 - 7 Num 4`	*reduce*
8.	×	`$ 0 Expr 1 - 7 Factor 3`	*reduce*
9.	×	`$ 0 Expr 1 - 7 Term 11`	*shift*
10.	`Id`	`$ 0 Expr 1 - 7 Term 11 × 8`	*shift*
11.	`EOF`	`$ 0 Expr 1 - 7 Term 11 × 8 id 5`	*reduce*
12.	`EOF`	`$ 0 Expr 1 - 7 Term 11 × 8 Factor 12`	*reduce*
13.	`EOF`	`$ 0 Expr 1 - 7 Term 11`	*reduce*
14.	`EOF`	`$ 0 Expr 1`	*accept*

Figure 3.9: LR(1) parser states for `x − 2 × y`

2. reduce: shrinks the frontier by replacing the current handle with its right hand site. It discards all the intermediate states used to recognize the handle by popping off two stack items for each symbol in the handle's right-hand side. Next, it uses the state underneath the handle and the left-hand side to find a new recognizer state, and pushes both the left-hand side and the new state onto the stack.

3. accept: reports success back to the user. This state is only reached when the parser has reduced the frontier to the goal symbol and the lookahead character is `eof`. (All other entries in state s_1 indicate errors!)

4. error: reports a syntax error. This state is reached anytime the action table contains an entry other than shift, reduce, or accept. In the figure, these entries are left blank.

Entries in the ACTION table are encoded using the letter 's' as "shift" and 'r' as "reduce". Thus, the entry "s3" indicates the action "shift and go to state s_3", while "r5" indicates "reduce by production 5". On a reduce item, the new state is determined by the GOTO table entry for the left-hand side of the production, and the state revealed on top of the stack after the right hand side is popped.

To understand this parser, consider again our simple example. Figure 3.9 shows the succession of states that the LR(1) parser takes to parse the expression `x − 2 × y`. The parser shifts `id` onto the stack, then reduces it to a *Factor*, to a *Term*, and to an *Expr*. Next, it shifts the `-`, followed by the `Num`. At this point, it reduces `Num` to *Factor*, then *Factor* to *Term*. At this point, it shifts × and then `id` onto the stack. Finally, it reduces `id` to *Factor*, *Term* × *Factor* to *Term*, and then *Expr - Term* to *Expr*. At this point, with *Expr* on the stack and `eof` as the next token, it accepts the input string. This is similar to the sequence portrayed in Figure 3.5.

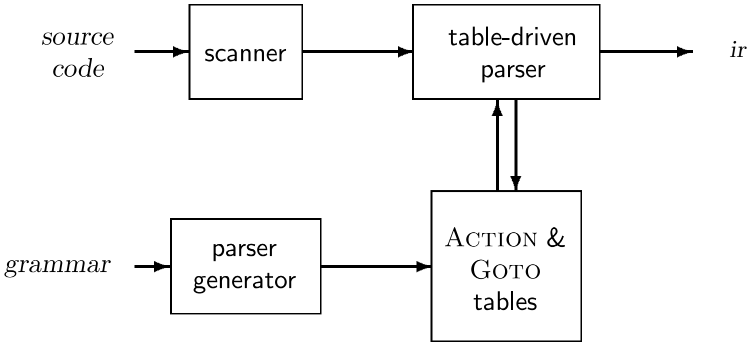


Figure 3.10: Structure of an LR(1) parser generator system

The key to building an LR(1) parser is constructing the ACTION and GOTO tables. These tables encode the actions of the handle-recognizing DFA, along with the information necessary to use limited right context to decide whether to shift, reduce, or accept. While it is possible to construct these tables by hand, the algorithm requires manipulation of lots of detail, as well as scrupulous book-keeping. Building LR(1) tables is a prime example of the kind of task that should be automated and relegated to a computer. Most LR(1) parse tables are constructed by parser generator systems, as shown in Figure 3.10. Using a parser generator, the compiler-writer creates a grammar that describes the source language; the parser generator converts that into ACTION and GOTO tables. In most such systems, the individual productions can be augmented with *ad hoc* code that will execute on each reduction. These "actions" are used for many purposes, including context-sensitive error checking, generating intermediate representations (see Section 4.4.3).

3.5.2 Building the tables

To construct ACTION and GOTO tables, an LR(1) parser generator builds a model of the handle-recognizing DFA and uses that model to fill in the tables. The model uses a set of LR(1) items to represent each parser state; these sets are constructed using a disciplined and systematic technique. The model is called the *canonical collection of sets of* LR(1) *items*. Each set in the canonical collection represents a parse state.

To explain the construction, we will use two examples. The SheepNoise grammar, *SN*, is small enough to use as a running example to clarify the individual steps in the process.

1.	*Goal*	→	*SheepNoise*
2.	*SheepNoise*	→	*SheepNoise* <u>`baa`</u>
3.		|	<u>`baa`</u>

This version of *SN* includes a distinct *Goal* production. Including a separate production for the goal symbol simplifies the implementation of the parser generator. As a second example, we will use the classic expression grammar. It includes complexity not found in *SN*; this makes it an interesting example, but too large to include incrementally in the text. Thus, this subsection ends with the classic expression grammar as a detailed example.

LR(1) Items An LR(1) item is a pair $[\alpha\rightarrow\beta \bullet \gamma\delta, \mathtt{a}]$, where $\alpha\rightarrow\beta\gamma\delta \in P$ is production of the grammar, the symbol $\bullet$ is a placeholder in the right-hand side, and $\mathtt{a} \in T$ is a word in the source language. Individual LR(1) items describe configurations of a bottom-up parser; they represent potential handles that would be consistent with the left context that the parser has already seen. The $\bullet$ marks the point in the production where the current upper frontier of the parse tree ends. (Alternatively, it marks the top of the parser's stack. These two views are functionally equivalent.)

For a production $\alpha\rightarrow\beta\gamma\delta$ and a lookahead symbol $\mathtt{a} \in T$, the addition of the placeholder generates four possible items, each with its own interpretation. In each case, the presence of the item in the set associated with some parser state indicates that the input that the parser has seen is consistent with the occurrence of an α followed by an $\mathtt{a}$. The position of $\bullet$ in the item provides more information.

$[\alpha\rightarrow\bullet\beta\gamma\delta, \mathtt{a}]$ indicates that an α would be valid and that recognizing a β at this point would be one step toward discovering an α.

$[\alpha\rightarrow\beta \bullet \gamma\delta, \mathtt{a}]$ indicates that the parser has progressed from the state where an α would be valid by recognizing β. The β is consistent with recognizing an α. The next step would be to recognize a γ.

$[\alpha\rightarrow\beta\gamma \bullet \delta, \mathtt{a}]$ indicates that the parser has moved forward from the state where α would be valid by recognizing $\beta\gamma$. At this point, recognizing a δ, followed by $\mathtt{a}$, would allow the parser to reduce $\beta\gamma\delta$ to α.

$[\alpha\rightarrow\beta\gamma\delta\bullet, \mathtt{a}]$ indicates that the parser has found $\beta\gamma\delta$ in a context where an α followed by $\mathtt{a}$ would be valid. If the lookahead symbol is $\mathtt{a}$, the parser can reduce $\beta\gamma\delta$ to α (and the item is a handle).

The lookahead symbols in LR(1) items encode left-context that the parser has already seen, in the sense that $[\alpha\rightarrow\beta\gamma\delta\bullet, \mathtt{a}]$ only indicates a reduction if the lookahead symbol is $\mathtt{a}$.

The SheepNoise grammar produces the following LR(1) items:

$[\mathit{Goal} \rightarrow \bullet\ \mathit{SheepNoise}, \mathtt{EOF}]$
$[\mathit{Goal} \rightarrow \mathit{SheepNoise} \bullet, \mathtt{EOF}]$
$[\mathit{SheepNoise} \rightarrow \bullet\ \mathtt{baa}, \mathtt{EOF}]$
$[\mathit{SheepNoise} \rightarrow \mathtt{baa} \bullet, \mathtt{EOF}]$
$[\mathit{SheepNoise} \rightarrow \bullet\ \mathtt{baa}, \mathtt{baa}]$
$[\mathit{SheepNoise} \rightarrow \mathtt{baa} \bullet, \mathtt{baa}]$
$[\mathit{SheepNoise} \rightarrow \bullet\ \mathit{SheepNoise}\ \mathtt{baa}, \mathtt{EOF}]$
$[\mathit{SheepNoise} \rightarrow \mathit{SheepNoise} \bullet \mathtt{baa}, \mathtt{EOF}]$
$[\mathit{SheepNoise} \rightarrow \mathit{SheepNoise}\ \mathtt{baa} \bullet, \mathtt{EOF}]$
$[\mathit{SheepNoise} \rightarrow \bullet\ \mathit{SheepNoise}\ \mathtt{baa}, \mathtt{baa}]$
$[\mathit{SheepNoise} \rightarrow \mathit{SheepNoise} \bullet \mathtt{baa}, \mathtt{baa}]$
$[\mathit{SheepNoise} \rightarrow \mathit{SheepNoise}\ \mathtt{baa} \bullet, \mathtt{baa}]$

```
for each $\alpha \in T$
    $\textsc{First}(\alpha) \leftarrow \alpha$
for each $\alpha \in NT$
    $\textsc{First}(\alpha) \leftarrow \emptyset$

while ($\textsc{First}$ sets are still changing)
    for each $p \in P$, where $p$ has the form $\alpha \rightarrow \beta$
        if $\beta$ is $\epsilon$
            then $\textsc{First}(\alpha) \leftarrow \textsc{First}(\alpha) \cup \{\epsilon\}$
        else if $\beta$ is $\beta_1\beta_2 \ldots \beta_k$ then
            $\textsc{First}(\alpha) \leftarrow \textsc{First}(\alpha) \cup \textsc{First}(\beta_1)$
            for i $\leftarrow$ 1 to $k-1$ by 1
                if $\epsilon \in \textsc{First}(\beta_i)$
                    then $\textsc{First}(\alpha) \leftarrow \textsc{First}(\alpha) \cup \textsc{First}(\beta_{i+1})$
                    else break
```

Figure 3.11: Computing FIRST sets

SN generates two terminal symbols. The first, `baa`, comes directly from the grammar. The second, `EOF` (for end of file) arises from the need to represent the parser's final state. The item [*Goal* → *SheepNoise* •, `EOF`] represents a parser configuration where it has already recognized a string that reduces to *Goal*, and it has exhausted the input, indicated by the lookahead of `EOF`.

Constructing FIRST *Sets* The construction of the canonical collection of sets of LR(1) items uses the FIRST sets for various grammar symbols. This set was informally defined in our discussion of predictive parsing (see Section 3.3.3). For the LR(1) construction, we need a more constructive definition:

$$\text{if } \alpha \ \Rightarrow^* \ \mathtt{a}\beta,\ \mathtt{a} \in T,\ \beta \in (T \cup NT)^* \text{ then } \mathtt{a} \in \textsc{First}(\alpha)$$

$$\text{if } \alpha \ \Rightarrow^* \ \epsilon \text{ then } \epsilon \in \textsc{First}(\alpha)$$

To compute FIRST sets, we apply these two rules inside a fixed-point framework. Figure 3.11 shows an algorithm that computes the FIRST set for every symbol in a context-free grammar $G = (T, NT, S, P)$.

On successive iterations, FIRST(α) takes on some value in $2^{(T \cup \epsilon)}$. The entire collection of FIRST sets is a subset of $2^{(T \cup \epsilon)}$. Each iteration of the while loop either increases the size of some FIRST set, or it changes no set. The algorithm halts when an iteration leaves all the FIRST sets unchanged. The actual running time of the algorithm depends on the structure of the grammar.

The FIRST sets for the augmented *SN* grammar are trivial.

Symbol	FIRST
Goal	`baa`
SheepNoise	`baa`
`baa`	`baa`
`EOF`	`EOF`

A more involved example of the FIRST set computation occurs with the classic expression grammar. (See the detailed example later in this section.)

Constructing the Canonical Collection The construction begins by building a model of the parser's initial state. To the grammar, $G = (T, NT, S, P)$, the construction adds an additional non-terminal, S', and one production, $S' \rightarrow S$. Adding this production allows us to specify the parser's start state concisely; it is represented by the LR(1) item $[S' \rightarrow \bullet S, \texttt{eof}]$.

Closure To this initial item, the construction needs to add all of the items implied by $[S' \rightarrow \bullet S, \texttt{eof}]$. The procedure `closure` does this.

```
closure($s_i$)
  while ($s_i$ is still changing)
    ∀ item $[\alpha \rightarrow \beta \bullet \gamma\delta, \mathsf{a}] \in s_i$
      ∀ production $\gamma \rightarrow \varsigma \in P$
        ∀ $\mathtt{b} \in$ FIRST($\delta\mathtt{a}$)
          if $[\gamma \rightarrow \bullet\varsigma, \mathsf{b}] \notin s_i$
             then add $[\gamma \rightarrow \bullet\varsigma, \mathsf{b}]$ to $s_i$
```

It iterates over the items in set. If an item has the • immediately before some non-terminal γ, it adds every production that can derive a γ, with the • at the left end of the right-hand side. The rationale is clear. If $[\alpha \rightarrow \beta \bullet \gamma\delta, \mathsf{a}]$ is a member of the set, then one potential completion for the left context is to find the string $\gamma\delta\mathsf{a}$, possibly followed by more context. This implies that γ is legal; hence, every production deriving a γ is part of a potential future handle. To complete the item, `closure` needs to add the appropriate lookahead symbol. If δ is non-empty, it generates items for every element of FIRST($\delta\mathsf{a}$). If δ is ϵ, this devolves into First(a) = a.

The `closure` computation is another fixed-point computation. At each point, the triply-nested loop either adds some elements to s_i or leaves it intact. Since the set of LR(1) items is finite, this loop must halt. In fact, $s_i \subseteq 2^{\text{ITEMS}}$, the power set of the set of all LR(1) items. The triply-nested loop looks expensive. However, close examination should convince you that each item in s_i must be processed exactly once. Only items added in the previous iteration need be processed in the inner two loops. The middle loop iterates over the set of alternative right-hand sides for a single production; this can easily be restricted so that each left-hand side is processed once per invocation of `closure`. The amount of work in the inner loop depends entirely on the size of First(δ`a`); if δ is ϵ, the loop makes a single trip for `a`. Thus, this computation may be much more sparse than it first appears.

In *SN*, the item [*Goal* → • *SheepNoise*,`EOF`] represents the initial state of the parser. (The parser is looking for an input string that reduces to *SheepNoise*, followed by `EOF`.) Taking the closure of this initial state produces the set

	Item	*Reason*
1.	[*Goal* → • *SheepNoise*,`EOF`]	original item
2.	[*SheepNoise* → • *SheepNoise* `baa`, `EOF`]	from 1, δ`a` is "`EOF`"
3.	[*SheepNoise* → • `baa`,`EOF`]	from 1, δ`a` is "`EOF`"
4.	[*SheepNoise* → • *SheepNoise* `baa`, `baa`]	from 2, δ`a` is "`baa EOF`"
5.	[*SheepNoise* → • `baa`,`baa`]	from 2, δ`a` is "`baa EOF`"

Items two and three derive directly from the first item. The final two items derive from item two; their lookahead symbol is just `First(baa EOF)`. This set represents the initial state, s_0, of an LR(1) parser for *SN*.

Goto If `closure`([S'→•S, `eof`]) computes the initial state s_0, the remaining step in the construction is to derive, from s_0, the other parser states. To accomplish this, we compute the state that would arise if we recognized a grammar symbol X while in s_0. The procedure `goto` does this.

```
goto($s_i, x$)
  new ← ∅
  ∀ items $i$ ∈ $s_i$
    if $i$ is [$\alpha \rightarrow \beta \bullet x\delta$, a] then
      moved ← [$\alpha \rightarrow \beta x \bullet \delta$, a]
      new ← new ∪ moved
  return closure(new)
```

`Goto` takes two arguments, a set of LR(1) items s_i and a grammar symbol x. It iterates over the items in s_i. When it finds one where • immediately precedes x, it creates the item resulting from recognizing x by moving the • rightward past x. It finds all such items, then returns their `closure` to fill out the state.

To find all the states that can be derived directly from s_0, the algorithm iterates over $x \in (T \cup NT)$ and computes `Goto`(s_0, x). This produces all the sets that are one symbol away from s_0. To compute the full canonical collection, we simply iterate this process to a fixed point.

In *SN*, the set s_0 derived earlier represents the initial state of the parser. To build a representation of the parser's state after seeing an initial `baa`, the construction computes `Goto(`s_0`,baa)`. `Goto` creates two items:

	Item	*Reason*
1.	[*SheepNoise* → `baa` •,`EOF`]	from item 3 in s_0
2.	[*SheepNoise* → `baa` •,`baa`]	from item 5 in s_0

The final part of `goto` invokes `closure` on this set. It finds no items to add because • is at the end of the production in each item.

The Algorithm The algorithm for constructing the canonical collection of sets of LR(1) items uses `closure` and `goto` to derive the set S of all parser states reachable from s_0.

```
$CC_0 \leftarrow$ closure($[S' \rightarrow \bullet S, \mathsf{EOF}]$)
while(new sets are still being added)
  for $S_j \in S$ and $x \in (T \cup NT)$
    if Goto($S_j, x$) is a new set, add it to $S$
      and record the transition
```

It begins by initializing S to contain s_0. Next, it systematically extends S by looking for any transition from a state in S to a state outside S. It does this constructively, by building all of the possible new states and checking them for membership in S.

Like the other computations in the construction, this is a fixed-point computation. The canonical collection, S, is a subset of 2^{ITEMS}. Each iteration of the while loop is monotonic; it can only add sets to S. Since S can grow no bigger than 2^{ITEMS}, the computation must halt. As in `closure`, a worklist implementation can avoid much of the work implied by the statement of the algorithm. Each trip through the while loop need only consider sets s_j added during the previous iteration. To further speed the process, it can inspect the items in s_j and only compute `goto`(s_j, x) for symbols that appear immediately after the $\bullet$ in some item in s_j. Taken together, these improvements should significantly decrease the amount of work required by the computation.

For *SN*, the computation proceeds as follows:

CC_0 is computed as *closure*([*Goal* → • *SheepNoise*, `EOF`]):
[*Goal* → • *SheepNoise*, `EOF`] [*SheepNoise* → • *SheepNoise baa*, `EOF`]
[*SheepNoise* → • *SheepNoise baa*, `baa`] [*SheepNoise* → • *baa*, `EOF`]
[*SheepNoise* → • *baa*, `baa`]

The first iteration produces two sets:

goto(CC_0,SheepNoise) is CC_1: [*Goal* → *SheepNoise* •, `EOF`]
[*SheepNoise* → *SheepNoise* • *baa*, `EOF`] [*SheepNoise* → *SheepNoise* • *baa*, `baa`]

goto(CC_0,baa) is CC_2: [*SheepNoise* → *baa* •, `EOF`] [*SheepNoise* → *baa* •, `baa`]

The second iteration produces one more set:

goto(CC_1,baa) is CC_3: [*SheepNoise* → *SheepNoise baa* •, `EOF`]
[*SheepNoise* → *SheepNoise baa* •, `baa`]

The final iteration produces no additional sets, so the computation terminates.

Filling in the Tables Given S, the canonical collection of sets of LR(1) items, the parser generator fills in the ACTION and GOTO tables by iterating through S and examining the items in each set $s_j \in S$. Each set s_j becomes a parser state. Its items generate the non-empty elements of one column of ACTION; the corresponding transitions recorded during construction of S specify the non-empty elements of GOTO. Three cases generate entries in the ACTION table:

```
$\forall\ s_i \in S$
  $\forall$ item $i \in s_i$
    if $i$ is $[\alpha \rightarrow \beta \bullet \mathtt{a}\gamma, \mathtt{b}]$ and goto$(s_i, \mathtt{a}) = s_j$, $\mathtt{a} \in T$
      then set $\text{ACTION}[i,a]$ to "shift j"
    else if $i$ is $[\alpha \rightarrow \beta\bullet, \mathtt{a}]$
      then set $\text{ACTION}[i,a]$ to "reduce $\alpha \rightarrow \beta$"
    else if $i$ is $[S' \rightarrow S\bullet, \mathtt{EOF}]$
      then set $\text{ACTION}[i,\mathtt{EOF}]$ to "accept"
  $\forall\ n \in NT$
    If goto$(s_i, A) = s_j$
      then set $\text{GOTO}[i, A]$ to $j$
```

Figure 3.12: LR(1) table-filling algorithm

1. An item of the form $[\alpha\rightarrow\beta\bullet\mathtt{b}\gamma, \mathtt{a}]$ indicates that encountering the terminal symbol b would be a valid next step toward discovering the non-terminal α. Thus, it generates a *shift* item on b in the current state. The next state for the recognizer is the state generated by computing *goto* on the current state with the terminal b. Either β or γ can be ϵ.

2. An item of the form $[\alpha\rightarrow\beta\bullet, \mathtt{a}]$ indicates that the parser has recognized a β, and if the lookahead is a, then the item is a handle. Thus, it generates a *reduce* item for the production $\alpha\rightarrow\beta$ on a in the current state.

3. The item $[S'\rightarrow S\bullet, \mathtt{eof}]$ is unique. It indicates the accepting state for the parser; the parser has recognized an input stream that reduces to the goal symbol. If the lookahead symbol is eof, then it satisfies the other acceptance criterion—it has consumed all the input. Thus, this item generates an *accept* action on eof in the current state.

The code in Figure 3.12 makes this concrete. For an LR(1) grammar, these items should uniquely define the non-error entries in the ACTION and GOTO tables.

Notice that the table-filling algorithm only essentially ignores items where the $\bullet$ precedes a non-terminal symbol. Shift actions are generated when $\bullet$ precedes a terminal. Reduce and accept actions are generated when $\bullet$ is at the right end of the production. What if s_i contains an item $[\alpha\rightarrow\beta\bullet\gamma\delta, \mathtt{a}]$, where $\gamma \in NT$? While this item does not generate any table entries itself, the items that closure derives from $[\alpha\rightarrow\beta\bullet\gamma\delta, \mathtt{a}]$ must include some of the form $[\nu\rightarrow\bullet b, \mathtt{c}]$, with $b \in T$. Thus, one effect of closure is to directly instantiate the FIRST set of γ into s_i. It chases down through the grammar until it finds each member of FIRST(γ) and puts the appropriate items into s_i to generate shift items for every $x \in$ FIRST(γ).

For our continuing example, the table-filling algorithm produces these two tables:

Action Table		
State	EOF	baa
0		shift 2
1	accept	shift 3
2	reduce 3	reduce 3
3	reduce 2	reduce 2

Goto Table	
State	SheepNoise
0	1
1	0
2	0
3	0

At this point, the tables can be used with the skeleton parser in Figure 3.8 to create an LR(1) parser for *SN*.

Errors in the Process If the grammar is not LR(1), the construction will attempt to multiply define one or more entries in the Action table. Two kinds of conflicts occur:

shift/reduce This conflict arises when some state s_i contains a pair of items $[\alpha \rightarrow \beta \bullet \mathtt{a}\gamma, \mathtt{b}]$ and $[\delta \rightarrow \nu \bullet, \mathtt{a}]$. The first item implies a *shift* on `a`, while the second implies a reduction by $\delta \rightarrow [\,]\nu$. Clearly, the parser cannot do both. In general, this conflict arises from an ambiguity like the `if-then-else` ambiguity. (See Section 3.2.3.)

reduce/reduce This conflict arises when some state s_i contains both $[\alpha \rightarrow \gamma \bullet, \mathtt{a}]$ and $[\beta \rightarrow \gamma \bullet, \mathtt{a}]$. The former implies that the parser should reduce γ to α, while the latter implies that it should reduce the same γ to β. In general, this conflict arises when the grammar contains two productions that have the same right-hand side, different left-hand sides, and allows them both to occur in the same place. (See Section 3.6.1.)

Typically, when a parser generator encounters one of these conflicts, it reports the error to the user and fails. The compiler-writer needs to resolve the ambiguity, as discussed elsewhere in this chapter, and try again.

The parser-generator can resolve a shift-reduce conflict in favor of shifting; this causes the parser to always favor the longer production over the shorter one. A better resolution, however, it to disambiguate the grammar.

Detailed Example To make this discussion more concrete, consider the classic expression grammar, augmented with a production *Goal*→*Expr*.

$$
\begin{array}{r|lcl}
1. & \mathit{Goal} & \rightarrow & \mathit{Expr} \\
2. & \mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
3. & & | & \mathit{Expr} - \mathit{Term} \\
4. & & | & \mathit{Term} \\
5. & \mathit{Term} & \rightarrow & \mathit{Term} \times \mathit{Factor} \\
6. & & | & \mathit{Term} \div \mathit{Factor} \\
7. & & | & \mathit{Factor} \\
8. & \mathit{Factor} & \rightarrow & (\ \mathit{Expr}\ ) \\
9. & & | & \mathtt{Num} \\
10. & & | & \mathtt{Id}
\end{array}
$$

Notice the productions have been renumbered. (Production numbers show up in "reduce" entries in the Action table.)

The FIRST sets for the augmented grammar are as follows:

	FIRST
Goal	(, Num, Id
Expr	(, Num, Id
Term	(, Num, Id
Factor	(, Num, Id
Num	Num
Id	Id

	FIRST
+	+
−	−
×	×
÷	÷
(	(
)	)

The initial step in constructing the Canonical Collection of Sets of LR(1) Items forms an initial item, [*Goal* → • *Expr*,EOF] and takes its closure to produce the first set.

CC$_0$: [*Goal* → • *Expr*,EOF], [*Expr* → • *Expr* + *Term*,{EOF,+,-}],
[*Expr* → • *Expr* - *Term*,{EOF,+,-}], [*Expr* → • *Term*,{EOF,+,-}],
[*Term* → • *Term* × *Factor*,{EOF,+,-,×,÷}],
[*Term* → • *Term* ÷ *Factor*,{EOF,+,-,×,÷}],
[*Term* → • *Factor*,{EOF,+,-,×,÷}], [*Factor* → • (*Expr*),{EOF,+,-,×,÷}],
[*Factor* → • Num,{EOF,+,-,×,÷}], [*Factor* → • Id,{EOF,+,-,×,÷}],

The first iteration computes ***goto*** on CC$_0$ and each symbol in the grammar. It produces six new sets, designating them CC$_1$ through CC$_6$.

CC$_1$: [*Goal* → *Expr* •,EOF], [*Expr* → *Expr* • + *Term*,{EOF,+,-}],
[*Expr* → *Expr* • - *Term*,{EOF,+,-}],

CC$_2$: [*Expr* → *Term* •,{EOF,+,-}], [*Term* → *Term* • × *Factor*,{EOF,+,-,×,÷}],
[*Term* → *Term* • ÷ *Factor*,{EOF,+,-,×,÷}],

CC$_3$: [*Term* → *Factor* •,{EOF,+,-,×,÷}],

CC$_4$: [*Expr* → • *Expr* + *Term*,{+,-,)}], [*Expr* → • *Expr* - *Term*,{+,-,)}],
[*Expr* → • *Term*,{+,-,)}], [*Term* → • *Term* × *Factor*,{+,-,×,÷,)}],
[*Term* → • *Term* ÷ *Factor*,{+,-,×,÷,)}], [*Term* → • *Factor*,{+,-,×,÷,)}],
[*Factor* → • (*Expr*),{+,-,×,÷,)}], [*Factor* → (• *Expr*),{EOF,+,-,×,÷}],
[*Factor* → • Num,{+,-,×,÷,)}], [*Factor* → • Id,{+,-,×,÷,)}],

CC$_5$: [*Factor* → Num •,{EOF,+,-,×,÷}],

CC$_6$: [*Factor* → Id •,{EOF,+,-,×,÷}],

Iteration two examines sets from CC$_1$ through CC$_6$. This produces new sets labelled CC$_7$ through CC$_{16}$.

CC$_7$: [*Expr* → *Expr* + • *Term*,{EOF,+,-}],
[*Term* → • *Term* × *Factor*,{EOF,+,-,×,÷}],
[*Term* → • *Term* ÷ *Factor*,{EOF,+,-,×,÷}],
[*Term* → • *Factor*,{EOF,+,-,×,÷}], [*Factor* → • (*Expr*),{EOF,+,-,×,÷}],
[*Factor* → • Num,{EOF,+,-,×,÷}], [*Factor* → • Id,{EOF,+,-,×,÷}],

CC$_8$: $[Expr \rightarrow Expr\ \texttt{-} \bullet Term, \{\texttt{EOF},\texttt{+},\texttt{-}\}]$,
$[Term \rightarrow \bullet\ Term \times Factor, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,
$[Term \rightarrow \bullet\ Term \div Factor, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,
$[Term \rightarrow \bullet\ Factor, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$, $[Factor \rightarrow \bullet\ (\ Expr\ ), \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,
$[Factor \rightarrow \bullet\ \texttt{Num}, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$, $[Factor \rightarrow \bullet\ \texttt{Id}, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,

CC$_9$: $[Term \rightarrow Term \times \bullet\ Factor, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,
$[Factor \rightarrow \bullet\ (\ Expr\ ), \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$, $[Factor \rightarrow \bullet\ \texttt{Num}, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,
$[Factor \rightarrow \bullet\ \texttt{Id}, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,

CC$_{10}$: $[Term \rightarrow Term \div \bullet\ Factor, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,
$[Factor \rightarrow \bullet\ (\ Expr\ ), \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$, $[Factor \rightarrow \bullet\ \texttt{Num}, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,
$[Factor \rightarrow \bullet\ \texttt{Id}, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,

CC$_{11}$: $[Expr \rightarrow Expr \bullet + Term, \{\texttt{+},\texttt{-},\texttt{)}\}]$, $[Expr \rightarrow Expr \bullet\ \texttt{-}\ Term, \{\texttt{+},\texttt{-},\texttt{)}\}]$,
$[Factor \rightarrow (\ Expr \bullet\ ), \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,

CC$_{12}$: $[Expr \rightarrow Term\ \bullet, \{\texttt{+},\texttt{-},\texttt{)}\}]$, $[Term \rightarrow Term \bullet \times Factor, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[Term \rightarrow Term \bullet \div Factor, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC$_{13}$: $[Term \rightarrow Factor\ \bullet, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC$_{14}$: $[Expr \rightarrow \bullet\ Expr + Term, \{\texttt{+},\texttt{-},\texttt{)}\}]$, $[Expr \rightarrow \bullet\ Expr\ \texttt{-}\ Term, \{\texttt{+},\texttt{-},\texttt{)}\}]$,
$[Expr \rightarrow \bullet\ Term, \{\texttt{+},\texttt{-},\texttt{)}\}]$, $[Term \rightarrow \bullet\ Term \times Factor, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[Term \rightarrow \bullet\ Term \div Factor, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$, $[Term \rightarrow \bullet\ Factor, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[Factor \rightarrow \bullet\ (\ Expr\ ), \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$, $[Factor \rightarrow (\ \bullet\ Expr\ ), \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[Factor \rightarrow \bullet\ \texttt{Num}, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$, $[Factor \rightarrow \bullet\ \texttt{Id}, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC$_{15}$: $[Factor \rightarrow \texttt{Num}\ \bullet, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC$_{16}$: $[Factor \rightarrow \texttt{Id}\ \bullet, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

Iteration three processes sets from CC$_7$ through CC$_{16}$. This adds sets CC$_{16}$ through CC$_{26}$ to the Canonical Collection.

CC$_{17}$: $[Expr \rightarrow Expr + Term\ \bullet, \{\texttt{EOF},\texttt{+},\texttt{-}\}]$,
$[Term \rightarrow Term \bullet \times Factor, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,
$[Term \rightarrow Term \bullet \div Factor, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,

CC$_{18}$: $[Expr \rightarrow Expr\ \texttt{-}\ Term\ \bullet, \{\texttt{EOF},\texttt{+},\texttt{-}\}]$,
$[Term \rightarrow Term \bullet \times Factor, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,
$[Term \rightarrow Term \bullet \div Factor, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,

CC$_{19}$: $[Term \rightarrow Term \times Factor\ \bullet, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,

CC$_{20}$: $[Term \rightarrow Term \div Factor\ \bullet, \{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,

CC$_{21}$: $[Expr \rightarrow Expr + \bullet\ Term, \{\texttt{+},\texttt{-},\texttt{)}\}]$, $[Term \rightarrow \bullet\ Term \times Factor, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[Term \rightarrow \bullet\ Term \div Factor, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$, $[Term \rightarrow \bullet\ Factor, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[Factor \rightarrow \bullet\ (\ Expr\ ), \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$, $[Factor \rightarrow \bullet\ \texttt{Num}, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[Factor \rightarrow \bullet\ \texttt{Id}, \{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC_{22}: $[\mathit{Expr} \rightarrow \mathit{Expr}\ \text{-}\ \bullet\ \mathit{Term},\{\texttt{+},\texttt{-},\texttt{)}\}]$, $[\mathit{Term} \rightarrow \bullet\ \mathit{Term} \times \mathit{Factor},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[\mathit{Term} \rightarrow \bullet\ \mathit{Term} \div \mathit{Factor},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$, $[\mathit{Term} \rightarrow \bullet\ \mathit{Factor},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[\mathit{Factor} \rightarrow \bullet\ (\ \mathit{Expr}\ ),\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$, $[\mathit{Factor} \rightarrow \bullet\ \texttt{Num},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[\mathit{Factor} \rightarrow \bullet\ \texttt{Id},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC_{23}: $[\mathit{Factor} \rightarrow (\ \mathit{Expr}\ )\ \bullet,\{\texttt{EOF},\texttt{+},\texttt{-},\times,\div\}]$,

CC_{24}: $[\mathit{Term} \rightarrow \mathit{Term} \times \bullet\ \mathit{Factor},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[\mathit{Factor} \rightarrow \bullet\ (\ \mathit{Expr}\ ),\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$, $[\mathit{Factor} \rightarrow \bullet\ \texttt{Num},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[\mathit{Factor} \rightarrow \bullet\ \texttt{Id},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC_{25}: $[\mathit{Term} \rightarrow \mathit{Term} \div \bullet\ \mathit{Factor},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[\mathit{Factor} \rightarrow \bullet\ (\ \mathit{Expr}\ ),\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$, $[\mathit{Factor} \rightarrow \bullet\ \texttt{Num},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[\mathit{Factor} \rightarrow \bullet\ \texttt{Id},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC_{26}: $[\mathit{Expr} \rightarrow \mathit{Expr}\ \bullet + \mathit{Term},\{\texttt{+},\texttt{-},\texttt{)}\}]$, $[\mathit{Expr} \rightarrow \mathit{Expr}\ \bullet\ \text{-}\ \mathit{Term},\{\texttt{+},\texttt{-},\texttt{)}\}]$,
$[\mathit{Factor} \rightarrow (\ \mathit{Expr}\ \bullet\ ),\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

Iteration four looks at sets CC_{17} through CC_{26}. This adds five more sets to the Canonical Collection, labelled CC_{27} through CC_{31}.

CC_{27}: $[\mathit{Expr} \rightarrow \mathit{Expr} + \mathit{Term}\ \bullet,\{\texttt{+},\texttt{-},\texttt{)}\}]$, $[\mathit{Term} \rightarrow \mathit{Term}\ \bullet \times \mathit{Factor},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[\mathit{Term} \rightarrow \mathit{Term}\ \bullet \div \mathit{Factor},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC_{28}: $[\mathit{Expr} \rightarrow \mathit{Expr}\ \text{-}\ \mathit{Term}\ \bullet,\{\texttt{+},\texttt{-},\texttt{)}\}]$, $[\mathit{Term} \rightarrow \mathit{Term}\ \bullet \times \mathit{Factor},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,
$[\mathit{Term} \rightarrow \mathit{Term}\ \bullet \div \mathit{Factor},\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC_{29}: $[\mathit{Term} \rightarrow \mathit{Term} \times \mathit{Factor}\ \bullet,\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC_{30}: $[\mathit{Term} \rightarrow \mathit{Term} \div \mathit{Factor}\ \bullet,\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

CC_{31}: $[\mathit{Factor} \rightarrow (\ \mathit{Expr}\ )\ \bullet,\{\texttt{+},\texttt{-},\times,\div,\texttt{)}\}]$,

Iteration five finds that every set it examines is already in the Canonical Collection, so the algorithm has reached its fixed point and it halts. Applying the table construction algorithm from Figure 3.12 produces the Action table shown in Figure 3.13 and the Goto table shown in Figure 3.14.

3.5.3 Shrinking the Action and Goto Tables

As Figures 3.13 and 3.14 show, the LR(1) tables generated for relatively small grammars can be quite large. Many techniques exist for shrinking these tables. This section describes three approaches to reducing table size.

Combining Rows or Columns If the table generator can find two rows, or two columns, that are Identical, it can combine them. In Figure 3.13, the rows for states zero and seven through ten are identical, as are rows 4, 14, 21, 22, 24, and 25. The table generator can implement each of these sets once, and remap the states accordingly. This would remove five rows from the table, reducing its size by roughly fifteen percent. To use this table, the skeleton parser needs 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

Action Table									
State	EOF	+	-	×	÷	(	)	Num	Id
0						s 4		s 5	s 6
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
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
8						s 4		s 5	s 6
9						s 4		s 5	s 6
10						s 4		s 5	s 6
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
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
22						s 14		s 15	s 16
23	r 8	r 8	r 8	r 8	r 8				
24						s 14		s 15	s 16
25						s 14		s 15	s 16
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 3.13: Action table for the classic expression grammar

Goto Table			
State	*Expr*	*Term*	*Factor*
0	1	2	3
1			
2			
3			
4	11	12	13
5			
6			
7		17	3
8		18	3
9			19
10			20
11			
12			
13			
14	26	12	13
15			
16			
17			
18			
19			
20			
21		27	13
22		28	13
23			
24			29
25			30
26			
27			
28			
29			
30			
31			

Figure 3.14: Goto table for the classic expression grammar

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 discover two rows or two columns that differ only in cases where one of the two has an "error" entry (denoted by a blank in our figures). In Figure 3.13, the column for `EOF` and for `Num` differ only where one or the other has a blank. Combining these columns produces a table that has the same behavior on correct inputs. The error behavior of the parser will change; several authors have published studies that show when such columns can be combined without adversely affecting the parser's ability to detect errors.

Combining rows and columns produces 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 for representing sparse matrices—again, the implementor must consider the tradeoff of memory size against any increase in access costs.

Using Other Construction Algorithms Several other algorithms for constructing 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. These grammars are restricted so that the lookahead symbols in the LR(1) items are not needed. The algorithm uses a set, called the FOLLOW set, to distinguish between cases where the parser should shift and those where it should reduce. (The FOLLOW set contains all of the terminals that can appear immediately after some non-terminal α in the input. The algorithm for constructing follow sets is similar to the one for building FIRST sets.) In practice, this mechanism is powerful enough to resolve most grammars of practical interest. Because the algorithm uses LR(0 items, it constructs a smaller canonical collection and its table has correspondingly fewer rows.

The LALR(1) algorithm capitalizes on the observation that some items in the set representing a state are critical, and the remaining ones can be derived from the critical items. The table construction item only represents these critical items; again, this produces a smaller canonical collection. The table entries corresponding to non-critical items can be synthesized late in the process.

The 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. However, in a mildly counter-intuitive result, any language that has an LR(1) grammar also has an LALR(1) grammar and an SLR(1) grammar. The grammars for these more restrictive forms will be shaped in a way that allows their respective construction algorithms to resolve the difference between situations where the parser should shift and those where it should reduce.

Shrinking the Grammar In many cases, the compiler writer can recode the grammar to reduce the number of productions that it contains. This usually leads 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* → `Id` 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 `Id` into a single symbol, `Val`, removes a column from the action table. To make this work, in practice, the scanner must return the same syntactic category, or token, for both numbers and identifiers.

Similar arguments can be made for combining × and ÷ into a single terminal `MulDiv`, and for combining + and − into a single terminal `AddSub`. Each of these replacements removes a terminal symbol and a production. This shrinks the size of the Canonical Collection of Sets of LR(1) Items, which removes rows from the table. It reduces the number of columns as well.

These three changes produce the following reduced expression grammar:

1.	*Goal*	→	*Expr*
2.	*Expr*	→	*Expr* `AddSub` *Term*
3.		|	*Term*
4.	*Term*	→	*Term* `MulDiv` *Factor*
5.		|	*Factor*
6.	*Factor*	→	`(` *Expr* `)`
7.		|	`Val`

Action Table

	EOF	Add-Sub	Mul-Div	(	)	Val
0				s 4		s 5
1	acc	s 6				
2	r 3	r 3	s 7			
3	r 5	r 5	r 5			
4				s 11		s 12
5	r 7	r 7	r 7			
6				s 4		s 5
7				s 4		s 5
8		s 15			s 16	
9		r 3	s 17		r 3	
10		r 5	r 5		r 5	
11				s 11		s 12
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
16	r 6	r 6	r 6			
17				s 11		s 12
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	

Goto Table

	Expr	*Term*	*Factor*
0	1	2	3
1			
2			
3			
4	8	9	10
5			
6		13	3
7			14
8			
9			
10			
11	18	9	10
12			
13			
14			
15		19	10
16			
17			20
18			
19			
20			
21			

Figure 3.15: Tables for the reduced expression grammar

The resulting Action and Goto tables are shown in Figure 3.15. The Action table contains 126 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 forty-eight percent reduction in table size. Note, however, that the tables still contain duplicate rows, such as 0, 6, and 7 in the Action table, or rows 4, 11, 15, and 17 in the Action table, and all of the identical rows in the Goto table. If table size is a serious concern, these techniques should be used together.

Directly Encoding the Table As a final improvement, the parser generator can abandon completely the table-driven skeleton parser in favor of a hard-coded implementation. Each state becomes a small case statement or a collection of `if–then–else` statements that test the type of the next symbol and either shift, reduce, accept, or produce an error. The entire contents of the Action and Goto tables can be encoded in this way. (A similar transformation for scanners is discusses in Section 2.8.2.)

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 a larger code size, since each state now includes more 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 table-driven skeleton parser. With appropriate code layout techniques, the resulting parser can exhibit strong locality in both the instruction cache and the paging system, an arena where seemingly random accesses to large tables produces poor performance.

3.6 Practical Issues

3.6.1 Handling Context-Sensitive Ambiguity

A second type of ambiguity arises from overloading the meaning of a word in the language. One example of this problem arose in the definitions of Fortran and PL/I. Both these languages use the same characters to enclose the index expression of an array and the argument list of a subroutine or function. Given a textual reference, such as `foo(i,j)`, the compiler cannot tell if `foo` is a two-dimensional array or a procedure that must be invoked. Differentiating between these two cases requires knowledge of `foo`'s type. This information is not syntactically obvious. The scanner undoubtedly classifies `foo` as an `Identifier` in either case. A function call and an array reference can appear in many of the same situations.

Resolving this ambiguity requires extra-syntactic knowledge. We have used two different approaches to solve this problem over the years. First, the scanner can should classify identifiers based on their declared type, rather than their micro-syntactic properties. This requires some hand-shaking between the scanner and the parser; the coordination is not hard to arrange as long as the language has a define-before-use rule. Since the declaration is parsed before the use occurs, the parser can make its internal symbol table available to the scanner to resolve identifiers into distinct classes, like *variable-name* and *function-name*. This allows the scanner to return a token type that distinguishes between the function invocation and the array reference.

The alternative is to rewrite the grammar so that it recognizes both the function invocation and the array reference in a single production. In this scheme, the issue is deferred until a later step in translation, when it can be resolved with information from the declarations. The parser must construct a representation that preserves all the information needed by either resolution; the later step will then rewrite the reference into its appropriate form as an array reference or as a function invocation.

3.6.2 Optimizing a Grammar

In the parser for a typical Algol-like language, a large share of the time is spent parsing expressions. Consider, again, the classic expression grammar from Section 3.2.3.

$$
\begin{array}{r|lcl}
1. & \mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
2. & & \mid & \mathit{Expr} - \mathit{Term} \\
3. & & \mid & \mathit{Term} \\
4. & \mathit{Term} & \rightarrow & \mathit{Term} \times \mathit{Factor} \\
5. & & \mid & \mathit{Term} \div \mathit{Factor} \\
6. & & \mid & \mathit{Factor} \\
7. & \mathit{Factor} & \rightarrow & (\ \mathit{Expr}\ ) \\
8. & & \mid & \texttt{Num} \\
9. & & \mid & \texttt{Id}
\end{array}
$$

Each production causes some sequence of shift and reduce actions in the parser. Some of the productions exist for cosmetic reasons; we can transform the grammar to eliminate them and the corresponding reduce actions. (Shift operations cannot be eliminated; they correspond to words in the input stream. Shortening the grammar does not change the input!) For example, we can replace any occurrence of *Factor* on the right hand side of a production with each of the right hand sides for *Factor*.

$$
\begin{array}{lclclcl}
\mathit{Term} & \rightarrow & \mathit{Term} \times (\ \mathit{Expr}\ ) & \mid & \mathit{Term} \times \texttt{Id} & \mid & \mathit{Term} \times \texttt{Num} \\
& \mid & \mathit{Term} \div (\ \mathit{Expr}\ ) & \mid & \mathit{Term} \div \texttt{Id} & \mid & \mathit{Term} \div \texttt{Num} \\
& \mid & (\ \mathit{Expr}\ ) & \mid & \texttt{Id} & \mid & \texttt{Num}
\end{array}
$$

This increases the number of productions, but removes an additional reduce action in the parser. Similarly, productions that have only one symbol on their right hand side such as the production *Expr* → *Term*, can be eliminated by an appropriate forward substitution.[3]

3.6.3 Left versus Right Recursion

As we have seen, top-down parsers need right recursive grammars rather than left recursive grammars. Bottom-up parsers can accommodate either left recursion or right recursion. Thus, the compiler writer has a choice between left recursion and right recursion in laying out the grammar for a bottom-up parser. Several factors plays into this decision.

Stack Depth In general, left recursion can lead to smaller stack depths. Consider two alternative grammars for a simple list construct.

$$
\begin{array}{lcl}
\mathit{List} & \rightarrow & \mathit{List}\ \mathit{Elt} \\
& \mid & \mathit{Elt}
\end{array}
\qquad\qquad
\begin{array}{lcl}
\mathit{List} & \rightarrow & \mathit{Elt}\ \mathit{List} \\
& \mid & \mathit{Elt}
\end{array}
$$

Using each grammar to produce a list of five elements, we produce the following derivations.

[3] These productions are sometimes called *useless productions*. They serve a purpose—making the grammar more compact and, perhaps, more readable. They are not, however, strictly necessary.

$List$	$List$
$List\ Elt_5$	$Elt_1\ List$
$List\ Elt_4\ Elt_5$	$Elt_1\ Elt_2\ List$
$List\ Elt_3\ Elt_4\ Elt_5$	$Elt_1\ Elt_2\ Elt_3\ List$
$List\ Elt_2\ Elt_3\ Elt_4\ Elt_5$	$Elt_1\ Elt_2\ Elt_3\ Elt_4\ List$
$Elt_1\ Elt_2\ Elt_3\ Elt_4\ Elt_5$	$Elt_1\ Elt_2\ Elt_3\ Elt_4\ Elt_5$

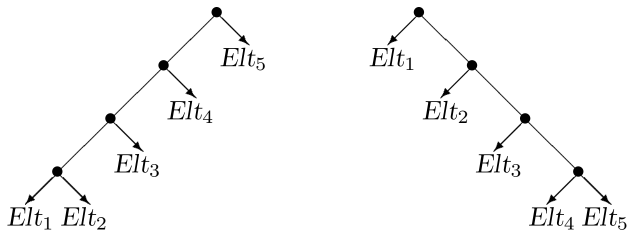
Since the parser constructs this sequence in reverse, reading the derivation from bottom line to top line allows us to follow the parser's actions.

- The left recursive grammar shifts Elt_1 onto its stack and immediately reduces it to *List*. Next, it shifts Elt_2 onto the stack and reduces it to *List*. It proceeds until it has shifted each of the five Elt_is onto the stack and reduced them to *List*. Thus, the stack reaches a maximum depth of two and an average depth of $\frac{10}{6} = 1\frac{2}{3}$.
- The right recursive version will shift all five Elt_i's onto its stack. Next, it reduces Elt_5 to *List* using rule two, and the remaining Elt_i's using rule one. Thus, its maximum stack depth will be five and its average will be $\frac{20}{6} = 3\frac{2}{3}$.

The right recursive grammar requires more stack space; in fact, its maximum stack depth is bounded only by the length of the list. In contrast, the maximum stack depth of the left recursive grammar is a function of the grammar rather than the input stream.

For short lists, this is not a problem. If, however, the list has hundreds of elements, the difference in space can be dramatic. If all other issues are equal, the smaller stack height is an advantage.

Associativity Left recursion naturally produces left associativity. Right recursion naturally produces right associativity. In some cases, the order of evaluation makes a difference. Consider the abstract syntax trees for the five element lists constructed earlier.



The left-recursive grammar reduces Elt_1 to a *List*, then reduces *List* Elt_1, and so on. This produces the AST shown on the left. Similarly, the right recursive grammar produces the AST on the right.

With a list, neither of these orders is obviously correct, although the right recursive AST may seem more natural. Consider, however, the result if we replace the list constructor with addition, as in the grammars

$$\begin{array}{lcl@{\qquad\qquad}lcl} \mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Operand} & \mathit{Expr} & \rightarrow & \mathit{Operand} + \mathit{Expr} \\ & | & \mathit{Operand} & & | & \mathit{Operand} \end{array}$$

Here, the difference between the ASTs for a five-element sum is obvious. The left-recursive grammar generates an AST that implies a left-to-right evaluation order, while the right recursive grammar generates a right-to-left evaluation order.

With some number systems, such as floating-point arithmetic on a computer, this reassociation can produce different results. Since floating-point arithmetic actually represents a small mantissa relative to the range of the exponent, addition becomes an identity operation for two numbers that are far enough apart in magnitude. If, for example, the processor's floating-precision is fourteen decimal digits, and $\mathit{Elt}_5 - \mathit{Elt}_4 > 10^{15}$, then the processor will compute $\mathit{Elt}_5 + \mathit{Elt}_4 = \mathit{Elt}_5$. If the other three values, Elt_1, Elt_2, and Elt_3 are also small relative to Elt_5, but their sum is large enough so that

$$\mathit{Elt}_5 - \sum_{i=1}^{4} \mathit{Elt}_i > 10^{14},$$

then left-to-right and right-to-left evaluation produce different answers.

The issue is not that one evaluation order is inherently correct while the other is wrong. The real issue is that the compiler must preserve the expected evaluation order. If the source language specifies an order for evaluating expressions, the compiler must ensure that the code it generates follows that order. The compiler writer can accomplish this in one of two ways: writing the expression grammar so that it produces the desired order, or taking care to generate the intermediate representation to reflect the opposite associativity, as described in Section 4.4.3.

3.7 Summary and Perspective

Almost every compiler contains a parser. For many years, parsing was a subject of intense interest. This led to the development of many different techniques for building efficient parsers. The LR(1) family of grammars includes all of the context-free grammars 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), LALR(1), and SLR(1) grammars, has decreased interest in other automatic parsing techniques (such as LL(1) and 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 for detecting and repairing syntax errors. The compiler writer can more easily finesse ambiguities in the source language that might trouble an LR(1) parser—such as, a language where keyword names can appear as identifiers. They are quite 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.) 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.

Questions

1. 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 have a table-generator handy.)

   Now, consider the same contrast for the `write` statement in Fortran 77. Its syntax is given by the following set of rules.

Chapter 4

Context-Sensitive Analysis

4.1 Introduction

Many of the important properties of a programming language cannot be specified in a context-free grammar. For example, to prepare a program for translation, the compiler needs to gather all of the information available to it for each variable used in the code. In many languages, this issue is addressed by a rule that requires a declaration for each variable before its use. To make checking this rule more efficient, the language might require that all declarations occur before any executable statements. The compiler must enforce these rules.

The compiler can use a syntactic mechanism to enforce the ordering of declarations and executables. A production such as

$$\textit{ProcedureBody} \rightarrow \textit{Declarations}\ \ \textit{Executables}$$

where the non-terminals have the obvious meanings, ensures that all *Declarations* occur before the *Executables*. A program that intermixes declarations with executable statements will raise a syntax error in the parser. However, this does nothing to check the deeper rule—that the program declares each variable before its first use in an executable statement.

Enforcing this second rule requires a deeper level of knowledge than can be encoded in the context-free grammar, which deals with syntactic categories rather than specific words. Thus, the grammar can specify the positions in an expression where a variable name can occur. The parser can recognize that the grammar allows the variable name to occur and it can tell that one has occurred. However, the grammar has no notation for matching up one instance of a variable name with another; that would require the grammar to specify a much deeper level of analysis.

Even though this rule is beyond the expressive power of a CFG, the compiler needs to enforce it. It must relate the use of `x` back to its declaration. The compiler needs an efficient mechanism to resolve this issue, and a host of others like it, that must be checked to ensure correctness.

4.2 The Problem

Before it can proceed with translation, the compiler must perform a number of computations that derive information about the code being compiled. For example, the compiler must record the basic information about the type, storage class, and dimension of each variable that its declaration contains, and it must use that information to check the type correctness of the various expressions and statements in the code. It must determine where to insert a conversion between data types, such as from a floating-point number to an integer.

These computations derive their basic facts from information that is implicit in the source program. The compiler uses the results to translate the code into another form. Thus, it is natural for these computations to follow the grammatical structure of the code. This chapter explores two techniques that tie the computation to the grammar used for parsing and use that structure to automate many of the details of the computation.

In Chapter 2, we saw that regular expressions provide most of the features necessary to automate the generation of scanners. The compiler writer specifies a regular expression and a token value for each syntactic category, and the tools produce a scanner. In Chapter 3, we saw that context-free grammars allow automation of much of the work required for parser construction; some of the rough edges must still be smoothed out by a skilled practitioner. The problems of context-sensitive analysis have not yet yielded to a neat, packaged solution.

This chapter presents two approaches to context-sensitive analysis: attribute grammars and *ad hoc syntax directed translation.*

- Attribute grammars provide a non-procedural formalism for associating computations with a context-free grammar. The definition of the attribute grammar consists of a set of equations; each equation is bound to a specific production. Tools exist for generating automatic evaluators from the set of rules. While attribute grammars have their strong points, they also have weaknesses that have prevented widespread adoption of the formalism for context-sensitive analysis.

- *Ad hoc* syntax-directed translation requires the compiler writer to produce small small snippets of code, called actions, that perform computations associated with a grammar. The technique draws on some of the insights that underlie attribute grammars to organize the computation, but allows the compiler writer to use arbitrary code in the actions. Despite the lack of formalism, *ad hoc* syntax directed translation remains the dominant technique used in compilers to perform context-sensitive analysis.

Formalism succeeded in both scanning and parsing, to the point that most compiler writers elect to use scanner generators based on regular expressions and to use parser generators based on context-free grammars. It has not succeeded as well with context-sensitive analysis. By exploring the strengths and weaknesses of attribute grammars, this chapter lays the foundation for understanding the principles that underlie syntax-directed translation. Next, this chapter explores

	Production	Attribution Rules

1.	$Number \rightarrow Sign\ List$	$List.pos \leftarrow 0$
		if $Sign.neg$
		then $Number.val \leftarrow -\ List.val$
		else $Number.val \leftarrow List.val$

2.	$Sign \rightarrow$ `+`	$Sign.neg \leftarrow$ *false*
3.	$Sign \rightarrow$ `-`	$Sign.neg \leftarrow$ *true*
4.	$List \rightarrow Bit$	$Bit.pos \leftarrow List.pos$ $List.val \leftarrow Bit.val$
5.	$List_0 \rightarrow List_1\ Bit$	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
6.	$Bit \rightarrow 0$	$Bit.val \leftarrow 0$
7.	$Bit \rightarrow 1$	$Bit.val \leftarrow 2^{Bit.pos}$

Figure 4.1: Attribute grammar for signed binary numbers

the mechanisms needed to build translators that perform *ad hoc* syntax-directed translation and presents a series of examples to illustrate how such computations can be structured. Finally, it concludes with a brief discussion of typical questions that a compiler might try to answer during context-sensitive analysis.

4.3 Attribute Grammars

An attributed context-free grammar, or *attribute grammar*, consists of a context-free grammar, augmented by a set of rules that specify a computation. Each rule defines one value, or *attribute*, in terms of the values of other attributes. The rule associates the attribute with a specific grammar symbol; each instance of the grammar symbol that occurs in a parse tree has a corresponding instance of the attribute. Because of the relationship between attribute instances and nodes in the parse tree, implementations are often described as adding fields for the attributes to the nodes of the parse tree.

The simple example shown in Figure 4.1 makes some of these notions concrete. The grammar uses seven productions to describe the language of signed binary numbers. Its grammar has four non-terminals, *Number*, *Sign*, *List*, and *Bit*, and four terminals `+`, `-`, `0`, and `1`. This particular grammar only associates values with non-terminal symbols. It defines the following attributes: *Number*.`val`, *Sign*.`neg`, *List*.`val`, *List*.`pos`, *Bit*.`val`, and *Bit*.`pos`. Subscripts are added to grammar symbols when needed to disambiguate a rule; *i.e.*, the occurrences of *List* in production 5. Notice that values flow from the right-hand side to the left-hand side and *vice versa*.

Production 4 shows this quite clearly. The `pos` attribute of *bit* receives the value of *list*`.pos`. We call *bit*`.pos` an *inherited attribute* because its value is derived from an attribute of its parent (or its siblings) in the parse tree. The *val* attribute of `list` receives its value from *bit*`.val`. We call *list*`.val` a *synthesized attribute* because its value is derived from an attribute of its children in the tree.

Given a string in the context free grammar, the attribute rules evaluate to set *number*`.val` to the decimal value of the binary input string. For example, the string `-101` causes the attribution shown on the left side of Figure 4.2. Notice that `number.val` has the value `-5`.

To evaluate an instance of the attribute grammar, the attributes specified in the various rules are instantiated for each grammar symbol in the parse (or each node in the parse tree). Thus, each instance of a `List` node in the example has its own copy of both `val` and `pos`. Each rule implicitly defines a set of dependences; the attribute being defined depends on each argument to the rule. Taken over the entire parse tree, these dependences form an *attribute dependence graph*. Edges in the graph follow the flow of values in the evaluation of a rule; an edge from $node_i.field_j$ to $node_k.field_l$ indicates that the rule defining $node_k.field_l$ uses the value of $node_i.field_j$ as one of its inputs. The right side of Figure 4.2 shows the dependence graph induced by the parse tree for the string `-101`.

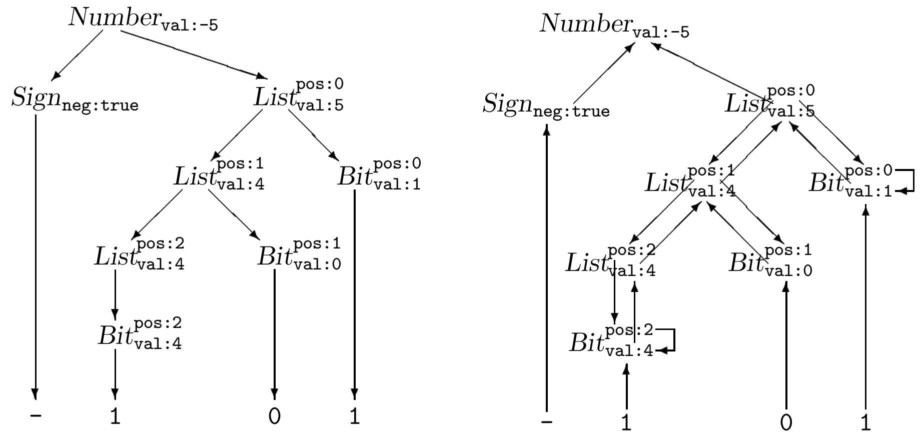
Any scheme for evaluating attributes must respect the relationships encoded implicitly in the attribute dependence graph. The rule that defines an attribute cannot be evaluated until all of the attribute values that it references have been defined. At that point, the value of the attribute is wholly defined. The attribute is immutable; its value cannot change. This can produce an evaluation order that is wholly unrelated to the order in which the rules appear in the grammar. For a production with three distinct rules, the middle one might evaluate long before the first or third. Evaluation order is determined by the dependences rather than any textual order.

To create and use an attribute grammar, the compiler writer determines a set of attributes for each terminal and non-terminal symbol in the grammar, and designs a set of rules to compute their values. Taken together, these specify a computation. To create an implementation, the compiler writer must create an evaluator; this can be done with an *ad hoc* program or by using an evaluator generator—the more attractive option. The evaluator generator takes as input the specification for the attribute grammar. It produces the code for an evaluator as its output. This is the attraction of an attribute grammar for the compiler writer; the tools take a high-level, non-procedural specification and automatically produce an implementation.

4.3.1 Evaluation Methods

The attribute grammar model has practical use only if we can build evaluators that interpret the rules to automatically evaluate an instance of the problem—a specific parse tree, for example. Many attribute evaluation techniques have been proposed in the literature. In general, they fall into three major categories.

Dynamic Methods These techniques use the structure of a particular instance of the attributed parse tree to determine the evaluation order. Knuth's original



Parse tree for `-101` *Dependence graph for* `-101`

Figure 4.2: Signed binary number example

paper on attribute grammars proposed an evaluator that operated in a manner similar to a *dataflow* architecture—each rule "fired" as soon as all its operands were available. In practical terms, this might be implemented using a queue of attributes that are ready for evaluation. As each attribute is evaluated, its successors in the attribute dependence graph are checked for "readiness" (see the description of "list scheduling" in Section 11.3).

A related scheme would build the attribute dependence graph, topologically sort it, and use the topological order to evaluate the attributes.

Oblivious Methods In these methods, the order of evaluation is independent of both the attribute grammar and the particular attributed parse tree. Presumably, the system's designer selects a method deemed appropriate for both the attribute grammar and the evaluation environment. Examples of this evaluation style include repeated left-to-right passes (until all attributes have values), repeated right-to-left passes, and alternating left-to-right and right-to-left passes. These methods have simple implementations and relatively small runtime overheads. They lack, of course, any improvement that can be derived from knowledge of the specific tree being attributed.

Rule-based Methods The rule-based methods rely on a static analysis of the attribute grammar to construct an evaluation order. In this framework, the evaluator relies on grammatical structure; thus, the parse tree guides the application of the rules. In the signed binary number example, the evaluation order for production four should use the first rule to set *Bit*.`pos`, recurse downward to *Bit*, and, on return, use *Bit*.`val` to set *List*.`val`. Similarly, for production five, it should evaluate the first two rules to define the `pos` attributes on the right hand side, then recurse downward to each child. On return, it can evaluate the third rule to set the *List*.`val` field of the parent *List* node. By performing the needed static analysis offline, at compiler-generation time, the evaluators built by these methods can be quite fast.

4.3.2 Circularity

If the attribute dependence graph contains a cycle, the tree cannot be completely attributed. A failure of this kind causes serious problems—for example, the compiler cannot generate code for its input. The catastrophic impact of cycles in the dependence graph suggests that the issue deserves close attention.

If a compiler uses attribute-grammar techniques, it must avoid creating circular attribute dependence graphs. Two approaches are possible.

- The compiler-writer can restrict the attribute grammar to a class that cannot give rise to circular dependence graphs. For example, restricting the grammar to use only synthesized attributes eliminates any possibility of a circular dependence graph. More general classes of non-circular attribute grammars exist; some, like *strongly-non-circular attribute grammars,* have polynomial-time tests for membership.

- The compiler-writer can design the attribute grammar so that it will not, on legal input programs, create a circular attribute dependence graph. The grammar might admit circularities, but some extra-grammatical constraint would prevent them. This might be a semantic constraint imposed with some other mechanism, or it might be a known convention that the input programs follow.

The rule-based evaluation methods may fail to construct an evaluator if the attribute grammar is circular. The oblivious methods and the dynamic methods will attempt to evaluate a circular dependence graph; they will simply fail to compute some of the attribute instances.

4.3.3 An Extended Example

To better understand the strengths and weaknesses of attribute grammars as a tool for specifying computations over the syntax of language, we will work through a more detailed example—estimating the execution time, in cycles, for a basic block.

A Simple Model Figure 4.3 shows a grammar that generates a sequence of assignment statements. The grammar is simplistic in that it allows only numbers and simple identifiers; nonetheless, it is complex enough to convey the complications that arise in estimating run-time behavior.

The right side of the figure shows a set of attribution rules that estimate the total cycle count for the block, assuming a single processor that executes one instruction at a time. The estimate appears in the `cost` attribute of the topmost *Block* node of the parse tree. The methodology is simple. Costs are computed bottom up; to read the example, start with the productions for *Factor*

Production			Attribution Rules
$Block_0$	→	$Block_1$ *Assign*	{ $Block_0$.*cost* ← $Block_1$.*cost* + *Assign*.*cost*; }
	|	*Assign*	{ *Block*.*cost* ← *Assign*.*cost*; }
Assign	→	*Ident* = *Expr* ;	{ *Assign*.*cost* ← *Cost(store)* + *Expr*.*cost*; }
$Expr_0$	→	$Expr_1$ + *Term*	{ $Expr_0$.*cost* ← $Expr_1$.*cost* + *Cost(add)* + *Term*.*cost*; }
	|	$Expr_1$ − *Term*	{ $Expr_0$.*cost* ← $Expr_1$.*cost* + *Cost(sub)* + *Term*.*cost*; }
	|	*Term*	{ *Expr*.*cost* ← *Term*.*cost*; }
$Term_0$	→	$Term_1$ × *Factor*	{ $Term_0$.*cost* ← $Term_1$.*cost* + *Cost(mult)* + *Factor*.*cost*; }
	|	$Term_1$ ÷ *Factor*	{ $Term_0$.*cost* ← $Term_1$.*cost* + *Cost(div)* + *Factor*.*cost*; }
	|	*Factor*	{ *Term*.*cost* ← *Factor*.*cost*; }
Factor	→	(*Expr*)	{ *Factor*.*cost* ← *Expr*.*cost*; }
	|	`Number`	{ *Factor*.*cost* ← *Cost(loadI)*; }
	|	`Ident`	{ *Factor*.*cost* ← *Cost(load)*; }

Figure 4.3: Simple attribute grammar for estimating execution time

and work your way up to the productions for *Block*. The function `COST` returns the latency of a given ILOC operation.

This attribute grammar uses only synthesized attributes—that is, all values flow in the direction from the leaves of the parse tree to its root. Such grammars are sometimes called *S-attributed grammars*. This style of attribution has a simple, rule-based evaluation scheme. It meshes well with bottom-up parsing; each rule can be evaluated when the parser reduces by the corresponding right-hand side. The attribute grammar approach appears to fit this problem well. The specification is short. It is easily understood. It leads to an efficient evaluator.

A More Accurate Model Unfortunately, this attribute grammar embodies a naive model for how the compiler handles variables. It assumes that each reference to an identifier generates a separate load operation. For the assignment `x = y + y;`, the model counts two load operations for `y`. Few compilers would generate a redundant load for `y`. More likely, the compiler would generate a

Production	Attribution Rules
Factor → `(` *Expr* `)`	{ *Factor.cost* ← *Expr.cost*; *Expr.Before* ← *Factor.Before*; *Factor.After* ← *Expr.After*; }

| `Number`	{ *Factor.cost* ← *Cost*(loadI);
	Factor.After ← *Factor.Before*;}

| `Ident`	{ *if* (`Ident`.*name* ∉ *Factor.Before*) *then* *Factor.cost* ← *Cost*(load); *Factor.After* ← *Factor.Before* ∪ `Ident`.*name*; *else* *Factor.cost* ← *0*; *Factor.After* ← *Factor.Before*; }

Figure 4.4: Rules for tracking loads in *Factor* productions

sequence such as

$$\begin{array}{lll} \texttt{loadAI} & \texttt{r}_0\texttt{,@y} & \Rightarrow \texttt{r}_y \\ \texttt{add} & \texttt{r}_y\texttt{,r}_y & \Rightarrow \texttt{r}_x \\ \texttt{storeAI} & \texttt{r}_x & \Rightarrow \texttt{r}_0\texttt{,@x} \end{array}$$

that loads `y` once. To approximate the compiler's behavior better, we can modify the attribute grammar to charge only a single load for each variable used in the block. This requires more complex attribution rules.

To account for loads more accurately, the rules must track references to each variable by the variable's name. These names are extra-grammatical, since the grammar tracks the syntactic category *Ident* rather than individual names such as `x`, `y`, and `z`. The rule for `Ident` should follow the general outline

```
if (Ident has not been loaded)
    then Factor.cost ← Cost(load);
    else Factor.cost ← 0;
```

The key to making this work is the test "`Ident` has not been loaded."

To implement this test, the compiler writer can add an attribute that holds the set of all variables already loaded. The production *Block* → *Assign* can initialize the set. The rules must thread the expression trees to pass the set through each assignment in the appropriate order. This suggests augmenting each node with a set *Before* and a set *After*; in practice, the sets are not necessary on leaves of the tree because rules for the leaf can reference the sets of its parent. The *Before* set for a node contains the names of all *Ident*s that occur earlier in the *Block*; each of these must have been loaded already. A node's *After* set contains all the names in its *Before* set, plus any Idents that would be loaded in the subtree rooted at that node.

The expanded rules for *Factor* are shown in Figure 4.4. The code assumes that each Ident has an attribute *name* containing its textual name. The first production, which derives (*Expr*), copies the *Before* set down into the *Expr* subtree and copies the resulting *After* set back up to the *Factor*. The second production, which derives Number, simply copies its parent's *Before* set into its parent's *After* set. Number must be a leaf in the tree; therefore, no further actions are needed. The final production, which derives Ident, performs the critical work. It tests the *Before* set to determine whether or not a load is needed and updates the parent's *cost* and *After* attributes accordingly.

To complete the specification, the compiler writer must add rules that copy the *Before* and *After* sets around the parse tree. These rules, sometimes called *copy rules*, connect the *Before* and *After* sets of the various *Factor* nodes. Because the attribution rules can only reference local attributes—defined as the attributes of a node's parent, its siblings, and its children—the attribute grammar must explicitly copy values around the grammar to ensure that they are local. Figure 4.5 shows the required rules for the other productions in the grammar. One additional rule has been added; it initializes the *Before* set of the first *Assign* statement to $\emptyset$.

This model is much more complex than the simple model. It has over three times as many rules; each rule must be written, understood, and evaluated. It uses both synthesized and inherited attributes; the simple bottom-up evaluation strategy will no longer work. Finally, the rules that manipulate the *Before* and *After* sets require a fair amount of attention—the kind of low-level detail that we would hope to avoid by using a system based on high-level specifications.

An Even More Complex Model As a final refinement, consider the impact of finite register sets on the model. The model used in the previous section assumes that the hardware provides an unlimited set of registers. In reality, computers provide finite, even small, register sets. To model the finite capacity of the register set, the compiler writer might limit the number of values allowed in the *Before* and *After* sets.

As a first step, we must replace the implementation of *Before* and *After* with a structure that holds exactly k values, where k is the size of the register set. Next, we can rewrite the rules for the production *Factor* → Ident to model register occupancy. If a value has not been loaded, and a register is available, it charges for a simple load. If a load is needed, but no register is available, it can evict a value from some register, and charge for the load. (Since the rule for *Assign* always charges for a store, the value in memory will be current. Thus, no store is needed when a value is evicted.) Finally, if the value has already been loaded and is still in a register, then no cost is charged.

This model complicates the rule set for *Factor* → Ident and requires a slightly more complex initial condition (in the rule for *Block* → *Assign*). It does not, however, complicate the copy rules for all of the other productions. Thus, the accuracy of the model does not add significantly to the complexity

Production	Attribution Rules
$Block_0 \rightarrow Block_1\ Assign$	$\{\ Block_0.\mathit{cost} \leftarrow Block_1.\mathit{cost}\ +$ $Assign.\mathit{cost};$ $Assign.\mathit{Before} \leftarrow Block_1.\mathit{After};$ $Block_0.\mathit{After} \leftarrow Assign.\mathit{After};\ \}$
$\mid\ Assign$	$\{\ Block.\mathit{cost} \leftarrow Assign.\mathit{cost};$ $Assign.\mathit{Before} \leftarrow \emptyset;$ $Block.\mathit{After} \leftarrow Assign.\mathit{After};\ \}$
$Assign \rightarrow Ident\ \mathtt{=}\ Expr;$	$\{\ Assign.\mathit{cost} \leftarrow \mathit{Cost}(\mathsf{store})\ +$ $Expr.\mathit{cost};$ $Expr.\mathit{Before} \leftarrow Assign.\mathit{Before};$ $Assign.\mathit{After} \leftarrow Expr.\mathit{After};\ \}$
$Expr_0 \rightarrow Expr_1 + Term$	$\{\ Expr_0.\mathit{cost} \leftarrow Expr_1.\mathit{cost}\ +$ $\mathit{Cost}(\mathsf{add}) + Term.\mathit{cost};$ $Expr_1.\mathit{Before} \leftarrow Expr_0.\mathit{Before};$ $Term.\mathit{Before} \leftarrow Expr_1.\mathit{After};$ $Expr_0.\mathit{After} \leftarrow Term.\mathit{After};\ \}$
$\mid\ Expr_1 - Term$	$\{\ Expr_0.\mathit{cost} \leftarrow Expr_1.\mathit{cost}\ +$ $\mathit{Cost}(\mathsf{sub}) + Term.\mathit{cost};$ $Expr_1.\mathit{Before} \leftarrow Expr_0.\mathit{Before};$ $Term.\mathit{Before} \leftarrow Expr_1.\mathit{After};$ $Expr_0.\mathit{After} \leftarrow Term.\mathit{After};\ \}$
$\mid\ Term$	$\{\ Expr.\mathit{cost} \leftarrow Term.\mathit{cost};$ $Term.\mathit{Before} \leftarrow Expr.\mathit{Before};$ $Expr.\mathit{After} \leftarrow Term.\mathit{After};\ \}$
$Term_0 \rightarrow Term_1 \times Factor$	$\{\ Term_0.\mathit{cost} \leftarrow Term_1.\mathit{cost}\ +$ $\mathit{Cost}(\mathsf{mult}) + Factor.\mathit{cost};$ $Term_1.\mathit{Before} \leftarrow Term_0.\mathit{Before};$ $Factor.\mathit{Before} \leftarrow Term_1.\mathit{After};$ $Term_0.\mathit{After} \leftarrow Factor.\mathit{After};\ \}$
$\mid\ Term_1 \div Factor$	$\{\ Term_0.\mathit{cost} \leftarrow Term_1.\mathit{cost}\ +$ $\mathit{Cost}(\mathsf{div}) + Factor.\mathit{cost};$ $Term_1.\mathit{Before} \leftarrow Term_0.\mathit{Before};$ $Factor.\mathit{Before} \leftarrow Term_1.\mathit{After};$ $Term_0.\mathit{After} \leftarrow Factor.\mathit{After};\ \}$
$\mid\ Factor$	$\{\ Term.\mathit{cost} \leftarrow Factor.\mathit{cost};$ $Factor.\mathit{Before} \leftarrow Term.\mathit{Before};$ $Term.\mathit{After} \leftarrow Factor.\mathit{After};\}$

Figure 4.5: Copy rules for tracking loads

of using an attribute grammar. All of the added complexity falls into the few rules that directly manipulate the model.

4.3.4 Problems with the Attribute Grammar Approach

The preceding example illustrates many of the computational issues that arise in using attribute grammars to perform context-sensitive computations on parse trees. Consider, for example, the "define before use" rule that requires a declaration for a variable before it can be used. This requires the attribute grammar to propagate information from declarations to uses. To accomplish this, the attribute grammar needs rules that pass declaration information upward in the parse tree to an ancestor that covers every executable statement that can refer to the variable. As the information passes up the tree, it must be aggregated into some larger structure that can hold multiple declarations. As the information flows down through the parse tree to the uses, it must be copied at each interior node. When the aggregated information reaches a use, the rule must find the relevant information in the aggregate structure and resolve the issue.

The structure of this solution is remarkably similar to that of our example. Information must be merged as it passes upward in the parse tree. Information must be copied around the parse tree from nodes that generate the information to nodes that need the information. Each of these rules must be specified. Copy rules can swell the size of an attribute grammar; compare Figure 4.3 against Figures 4.4 and 4.5. Furthermore, the evaluator executes each of these rules. When information is aggregated, as in the define-before-use rule or the framework for estimating execution times, a new copy of the information must be made each time that the rule changes its contents. Taken over the entire parse tree, this involves a significant amount of work and creates a large number of new attributes.

In a nutshell, solving non-local problems introduces significant overhead in terms of additional rules to copy values from node to node and in terms of space management to hold attribute values. These copy rules increase the amount of work that must be done during evaluation, albeit by a constant amount per node. They also make the attribute grammar itself larger—requiring the compiler writer to specify each copy rule. This adds another layer of work to the task of writing the attribute grammar.

As the number of attribute instances grows, the issue of storage management arises. With copy rules to enable non-local computations, the amount of attribute storage can increase significantly. The evaluator must manage attribute storage for both space and time; a poor storage management scheme can have a disproportionately large negative impact on the resource requirements of the evaluator.

The final problem with using an attribute grammar scheme to perform context-sensitive analysis is more subtle. The result of attribute evaluation is an attributed tree. The results of the analysis are distributed over that tree, in the form of attribute values. To use these results in later passes, the compiler must navigate the tree to locate the desired information. Lookups must traverse the tree to find the information, adding cost to the process. During the design process, the compiler-writer must plan where the information will be located at compile time and how to locate it efficiently.

One way to address all of these problems is to add a central repository for facts to the parser. In this scenario, an attribute rule can record information directly into a global table, where other rules can read it. This hybrid approach can eliminate many of the problems that arise from non-local information. Since the table can be accessed from any attribution rule, it has the effect of providing local access to any information already derived. This, in turn, introduces some implicit ordering constraints. For example, if the table includes information about the declared type and dimension of variables, the rules that enter this information into the table must execute before those that try to reference the information.[1] Introducing a central table for facts, however, begins to corrupt the purely functional nature of attribute specification.

It is not clear that attributed grammars are the right abstraction for performing the kinds of context-sensitive analysis that arise in a compiler. Advocates of attribute grammar techniques argue that all of the problems are manageable, and that the advantages of a high-level, non-procedural specification outweigh the problems. However, the attribute grammar approach has never achieved widespread popularity for a number of mundane reasons. Large problems, like the difficulty of performing non-local computation and the need to traverse the parse tree to discover answers to simple questions, have slowed the adoption of these ideas. Myriad small problems, such as space management for short-lived attributes, efficiency of evaluators, and the availability of high-quality, inexpensive tools, have also made these tools and techniques less attractive.

Still, the simplicity of the initial estimation model is attractive. If attribute flow can be constrained to a single direction, either synthesized or inherited, the resulting attribute grammar is simple and the evaluator is efficient. One example suggested by other authors is expression evaluation in a calculator or an interpreter. The flow of values follows the parse tree from leaves to root, so both the rules and the evaluator are straight forward. Similarly, applications that involve only local information often have good attribute grammar solutions. We will see an example of such a computation in Chapter 9, where we discuss instruction selection.

4.4 Ad-hoc Syntax-directed Translation

The rule-based evaluators for attribute grammars introduced a powerful idea that actually serves as the basis for the *ad hoc* techniques used for context-sensitive analysis in many compilers. In the rule-based evaluators, the compiler writer specifies a sequence of actions in terms of productions in the grammar. The underlying observation, that the actions required for context-sensitive analysis can be organized around the structure of the grammar, leads to a powerful, albeit *ad hoc*, approach to incorporating this kind of analysis into the process

[1] In fact, the copy rules in Figure 4.5 encode the same set of constraints. To see this clearly, draw the attribute dependence graph for an example.

of parsing a context-free grammar. We refer to this approach as *ad hoc* syntax-directed translation.

In this scheme, the compiler writer provides arbitrary snippets of code that will execute at parse time. Each snippet, or *action*, is directly tied to a production in the grammar. Each time the parser reduces by the right-hand side of some production, the corresponding action is invoked to perform its task. In a top-down, recursive-descent parser, the compiler writer simply adds the appropriate code to the parsing routines. The compiler writer has complete control over when the actions execute. In a shift-reduce parser, the actions are performed each time the parser performs a reduce action. This is more restrictive, but still workable.

The other points in the parse where the compiler writer might want to perform an action are: (1) in the middle of a production, or (2) on a shift action. To accomplish the first, the compiler writer can transform the grammar so that it reduces at the appropriate place. Usually, this involves breaking the production into two pieces around the point where the action should execute. A higher-level production is added that sequences the first part, then the second. When the first part reduces, the parser will invoke the action. To force actions on shifts, the compiler writer can either move them into the scanner, or add a production to hold the action. For example, to perform an action whenever the parser shifts terminal symbol *Variable*, the compiler writer can add a production

$$\textit{ShiftedVariable} \rightarrow \textit{Variable}$$

and replace every occurrence of *Variable* with *ShiftedVariable*. This adds an extra reduction for every terminal symbol. Thus, the additional cost is directly proportional to the number of terminal symbols in the program.

4.4.1 Making It Work

For *ad hoc* syntax-directed translation to work, the parser must provide mechanisms to sequence the application of the actions, to pass results between actions, and to provide convenient and consistent naming. We will describe these problems and their solution in shift-reduce parsers; analogous ideas will work for top-down parsers. `Yacc`, an early LR(1) parser generator for Unix systems, introduced a set of conventions to handle these problems. Most subsequent systems have used similar techniques.

Sequencing the Actions In fitting an *ad hoc* syntax directed translation scheme to a shift-reduce parser, the natural way to sequence actions is to associate each code snippet with the right-hand side of a production. When the parser reduces by that production, it invokes the code for the action. As discussed earlier, the compiler writer can massage the grammar to create additional reductions that will, in turn, invoke the code for their actions.

To execute the actions, we can make a minor modification to the skeleton LR(1) parser's reduce action (see Figure 3.8).

```
else if action[s,token] = "reduce $A \rightarrow \beta$" then
    invoke the appropriate reduce action
    pop $2 \times \mid \beta \mid$ symbols
    s ← top of stack
    push $A$
    push goto[s,$A$]
```

The parser generator can gather the syntax-directed actions together, embed them in a case statement that switches on the number of the production being reduced, and execute the case statement just before it pops the right-hand side from the stack.

Communicating Between Actions To connect the actions, the parser must provide a mechanism for passing values between the actions for related productions. Consider what happens in the execution time estimator when it recognizes the identifier `y` while parsing `x` ÷ `y`. The next two reductions are

$$\begin{array}{lll} \textit{Factor} & \rightarrow & \texttt{Ident} \\ \textit{Term} & \rightarrow & \textit{Term} \div \textit{Factor} \end{array}$$

For syntax-directed translation to work, the action associated with the first production, *Factor* → `Ident`, needs a location where it can store values. The action associated with the second production *Term* → *Term* ÷ *Factor* must know where to find the result of the action caused by reducing `x` to *Factor*.

The same mechanism must work with the other productions that can derive *Factor*, such as *Term* → *Term* × *Factor* and *Term* → *Factor*. For example, in `x` ÷ `y`, the values for the *Term* on the right hand side of *Term* → *Term* ÷ *Factor* is, itself, the result of an earlier reduction by *Factor* → `Ident`, followed by a reduction of *Term* → *Factor*. The lifetimes of the values produced by the action for *Factor* → `Ident` depend on the surrounding syntactic context; thus, the parser needs to manage the storage for values.

To accomplish this, a shift-reduce parser can simply store the results in the parsing stack. Each reduction pushes its result onto the stack. For the production *Term* → *Term* ÷ *Factor*, the topmost result will correspond to *Factor*. The second result will correspond to ÷, and the third result will correspond to *Term*. The results will be interspersed with grammar symbols and states, but they occur at fixed intervals in the stack. Any results that lie below the *Term*'s slot on the stack represent the results of other reductions that form a partial left context for the current reduction.

To add this behavior to the skeleton parser requires two further changes. To keep the changes simple, most parser generators restrict the results to a fixed size. Rather than popping $2 \times \mid \beta \mid$ symbols on a reduction by $A \rightarrow \beta$, it must now pop $3 \times \mid \beta \mid$ symbols. The result must be stacked in a consistent position; the simplest modification pushes the result before the grammar symbol. With these restrictions, the result that corresponds to each symbol on the right hand side can be easily found. When an action needs to return multiple values, or a complex value such as a piece of an abstract syntax tree, the action allocates a structure and pushes a pointer into the appropriate stack location.

Naming Values With this stack-based communication mechanism, the compiler writer needs a mechanism for naming the stack locations corresponding to symbols in the production's right-hand side. Yacc introduced a concise notation to address these problems. The symbol $$ refers to the result location for the current production. Thus, the assignment `$$ = 17;` would push the integer value seventeen as the result corresponding to the current reduction. For the right-hand side, the symbols $1, $2, …, $$n$ refer to the locations for the first, second, and n^{th} symbols in the right-hand side, respectively. These symbols translate directly into offsets from the top of the stack. $1 becomes $3 \times \mid \beta \mid$ slots below the top of the stack, while $4 becomes $3 \times (\mid \beta \mid -4+1)$ slots from the top of the stack. This simple, natural notation allows the action snippets to read and write the stack locations directly.

4.4.2 Back to the Example

To understand how *ad-hoc* syntax-directed translation works, consider rewriting the execution-time estimator using this approach. The primary drawback of the attribute grammar solution lies in the proliferation of rules to copy information around the tree. This creates many additional rules in the specification. It also creates many copies of the sets. Even a careful implementation that stores pointers to a single copy of each set instance must create new sets whenever an identifier's name is added to the set.

To address these problems in an *ad hoc* syntax-directed translation scheme, the compiler writer can introduce a central repository for information about variables, as suggested earlier. For example, the compiler can create a hash table that contains a record for each Ident in the code. If the compiler writer sets aside a field in the table, named InRegister, then the entire copy problem can be avoided. When the table is initialized, the InRegister field is set to false. The code for the production *Factor* → Ident checks the InRegister field and selects the appropriate cost for the reference to Ident. The code would look something like:

```
i = hash(Ident);
if (Table[i].Loaded ≠ true)
    then
        cost = cost + Cost(load);
        Table[i].Loaded = true;
```

Because the compiler writer can use arbitrary constructs, the cost can be accumulated into a single variable, rather than being passed around the parse tree. The resulting set of actions is somewhat smaller than the attribution rules for the simplest execution model, even though it can provide the accuracy of the more complex model. Figure 4.6 shows the full code for an *ad hoc* version of the example shown in Figures 4.4 and 4.5.

In the *ad hoc* version, several productions have no action. The remaining actions are quite simple, except for the action taken on reduction by Ident. All of the complication introduced by tracking loads falls into that single action;

Production			Syntax-directed Actions
$Block_0$	→	$Block_1$ *Assign*	
	|	*Assign*	{ `cost = 0;` }
Assign	→	*Ident* = *Expr* ;	{ *cost* = *cost* + *Cost*(`store`) }
$Expr_0$	→	$Expr_1$ + *Term*	{ *cost* = *cost* + *Cost*(`add`); }
	|	$Expr_1$ − *Term*	{ *cost* = *cost* + *cost*(`sub`); }
	|	*Term*	
$Term_0$	→	$Term_1$ × *Factor*	{ *cost* = *cost* + *Cost*(`mult`); }
	|	$Term_1$ ÷ *Factor*	{ *cost* = *cost* + *Cost*(`div`); }
	|	*Factor*	
Factor	→	(*Expr*)	
	|	`Number`	{ *cost* = *cost* + *Cost*(`loadI`); }

	|	`Ident`	{ *i* = *hash*(`Ident`);
			if (*Table[i].Loaded* ≠ *true*)
			then
			cost = *cost* + *Cost*(`load`);
			Table[i].Loaded = *true*; }

Figure 4.6: Tracking loads with *ad hoc* syntax-directed translation

contrast that with the attribute grammar version, where the task of passing around the *Before* and *After* sets came to dominate the specification. Because it can accumulate `cost` into a single variable and use a hash table to store global information, the *ad hoc* version is much cleaner and simpler. Of course, these same strategies could be applied in an attribute grammar framework, but doing so violates the spirit of the attribute grammar paradigm and forces all of the work outside the framework into an *ad hoc* setting.

4.4.3 Uses for Syntax-directed Translation

Compilers use syntax-directed translation schemes to perform many different tasks. By associating syntax-directed actions with the productions for source-language declarations, the compiler can accumulate information about the variables in a program and record it in a central repository—often called a *symbol table*. As part of the translation process, the compiler must discover the answer to many context-sensitive questions, similar to those mentioned in Section 4.1. Many, if not all, of these questions can be answered by placing the appropriate code in syntax-directed actions. The parser can build an IR for use by the rest of the compiler by systematic use of syntax-directed actions. Finally, syntax-directed translation can be used to perform complex analysis and transformations. To understand the varied uses of syntax-directed actions, we will examine several different applications of *ad hoc* syntax-directed translation.

Building a Symbol Table To centralize information about variables, labels, and procedures, most compilers construct a symbol table for the input program. The compiler writer can use syntax-directed actions to gather the information, insert that information into the symbol table, and perform any necessary processing. For example, the grammar fragment shown in Figure 4.7 describes a subset of the syntax for declaring variables in C. (It omits `typedef`s, `struct`s, `union`s, the type qualifiers `const` and `volatile`, and the initialization syntax; it also leaves several non-terminals unelaborated.) Consider the actions required to build symbol table entries for each declared variable.

Each *Declaration* begins with a set of one or more qualifiers that specify either the variable's type, its storage class, or both. The qualifiers are followed by a list of one or more variable names; the variable name can include a specification about indirection (one or more occurrences of `*`), about array dimensions, and about initial values for the variable. To build symbol table entries for each variable, the compiler writer can gather up the attributes from the qualifiers, add any indirection, dimension, or initialization attributes, and enter the variable in the table.

For example, to track storage classes, the parser might include a variable `StorageClass`, initializing it to a value "*none*." Each production that reduced to *StorageClass* would would set `StorageClass` to an appropriate value. The language definition allows either zero or one storage class specifier per declaration, even though the context-free grammar admits an arbitrary number. The syntax-directed action can easily check this condition. Thus, the following code might execute on a reduction by *StorageClass* → `register`:

```
if (StorageClass = none)
    then StorageClass = auto
    else report the error
```

Similar actions set `StorageClass` for the reductions by `static`, `extern`, and `register`. In the reduction for *DirectDeclarator* → `Identifier`, the action creates a new symbol table entry, and uses `StorageClass` to set the appropriate field in the symbol table. To complete the process, the action for the production *Declaration* → *SpecifierList InitDeclaratorList* needs to reset `StorageClass` to *none*.

Following these lines, the compiler writer can arrange to record all of the attributes for each variable. In the reduction to `Identifier`, this information can be written into the symbol table. Care must be taken to initialize and reset the attributes in the appropriate place—for example, the attribute set by the *Pointer* reduction should be reset for each *InitDeclarator*. The action routines can check for valid and invalid *TypeSpecifier* combinations, such as `signed char` (legal) and `double char` (illegal).

DeclarationList	$\rightarrow$	*DeclarationList Declaration*
	$\mid$	*Declaration*
Declaration	$\rightarrow$	*SpecifierList InitDeclaratorList* `;`
SpecifierList	$\rightarrow$	*Specifier SpecifierList*
	$\mid$	*Specifier*
Specifier	$\rightarrow$	*StorageClass*
	$\mid$	*TypeSpecifier*
StorageClass	$\rightarrow$	`auto`
	$\mid$	`static`
	$\mid$	`extern`
	$\mid$	`register`
TypeSpecifier	$\rightarrow$	`char`
	$\mid$	`short`
	$\mid$	`int`
	$\mid$	`long`
	$\mid$	`unsigned`
	$\mid$	`float`
	$\mid$	`double`
InitDeclaratorList	$\rightarrow$	*InitDeclaratorList, InitDeclarator*
	$\mid$	*InitDeclarator*
InitDeclarator	$\rightarrow$	*Declarator* = *Initializer*
	$\mid$	*Declarator*
Declarator	$\rightarrow$	*Pointer DirectDeclarator*
	$\mid$	*DirectDeclarator*
Pointer	$\rightarrow$	`*`
	$\mid$	`*` *Pointer*
DirectDeclarator	$\rightarrow$	`Identifier`
	$\mid$	`(` *Declarator* `)`
	$\mid$	*DirectDeclarator* `(` `)`
	$\mid$	*DirectDeclarator* `(` *ParameterTypeList* `)`
	$\mid$	*DirectDeclarator* `(` *IdentifierList* `)`
	$\mid$	*DirectDeclarator* `[` `]`
	$\mid$	*DirectDeclarator* `[` *ConstantExpr* `]`

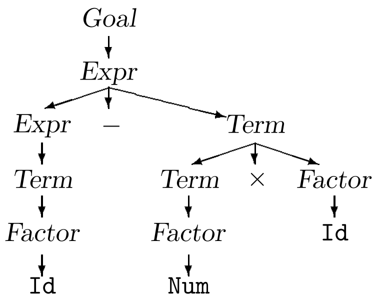
Figure 4.7: A subset of C's declaration syntax

When the parser finishes building the *DeclarationList*, it has symbol table entries for each variable declared in the current scope. At that point, the compiler may need to perform some housekeeping chores, such as assigning storage locations to declared variables. This can be done in an action for the production that leads to the *DeclarationList*. (That production has *DeclarationList* on its right-hand side, but not on its left-hand side.)

Building a Parse Tree Another major task that parsers often perform is building an intermediate representation for use by the rest of the compiler. Building a parse tree through syntax-directed actions is easy. Each time the parser reduces a production $A \rightarrow \beta\gamma\delta$, it should construct a node for A and make the nodes for β, γ, and δ children of A, in order. To accomplish this, it can push pointers to the appropriate nodes onto the stack.

In this scheme, each reduction creates a node to represent the non-terminal on its left-hand side. The node has a child for each grammar symbol on the right-hand side of the production. The syntax-directed action creates the new node, uses the pointers stored on the stack to connect the node to its children, and pushes the new node's address as its result.

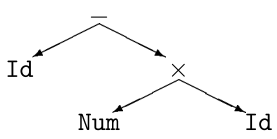
As an example, consider parsing `x - 2 × y` with the classic expression grammar. It produces the following parse tree:



Of course, the parse tree is large relative to its information content. The compiler writer might, instead, opt for an *abstract syntax tree* that retains the essential elements of the parse tree, but gets rid of internal nodes that add nothing to our understanding of the underlying code (see § 6.3.2).

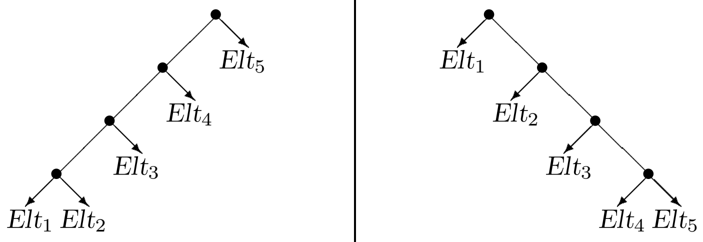
To build an abstract syntax tree, the parser follows the same general scheme as for a parse tree. However, it only builds the desired nodes. For a production like $A \rightarrow B$, the action returns as its result the pointer that corresponds to B. This eliminates many of the interior nodes. To further simplify the tree, the compiler writer can build a single node for a construct such as the `if–then–else`, rather than individual nodes for the `if`, the `then`, and the `else`.

An AST for `x - 2 × y` is much smaller than the corresponding parse tree:



Grammar	*Actions*
List → *List Elt*	*$$ ← L($1,$2);*
| *Elt*	*$$ ← $1;*

Grammar	*Actions*
List → *Elt List*	*$$ ← L($1,$2);*
| *Elt*	*$$ ← $1;*



Left Recursion

Right Recursion

Figure 4.8: Recursion versus Associativity

Changing Associativity As we saw in Section 3.6.3, associativity can make a difference in numerical computation. Similarly, it can change the way that data structures are built. We can use syntax-directed actions to build representations that reflect a different associativity than the grammar would naturally produce.

In general, left recursive grammars naturally produce left-associativity, while right-recursive grammars naturally produce right associativity. To see this consider the left-recursive and right-recursive list grammars, augmented with syntax-directed actions to build lists, shown at the top of Figure 4.8. Assume that `L(x,y)` is a constructor that returns a new node with `x` and `y` as its children. The lower part of the figure shows the result of applying the two translation schemes to an input consisting of five *Elt*s.

The two trees are, in many ways, equivalent. An in-order traversal of both trees visits the leaf nodes in the same order. However, the tree produced from the left recursive version is strangely counter-intuitive. If we add parentheses to reflect the tree structure, the left recursive tree is ((((Elt_1,Elt_2),Elt_3,)Elt_4), Elt_5) while the right recursive tree is (Elt_1,(Elt_2,(Elt_3,(Elt_4, Elt_5)))). The ordering produced by left recursion corresponds to the classic left-to-right ordering for algebraic operators. The ordering produced by right recursion corresponds to the notion of a list as introduced in programming languages like Lisp and Scheme.

Sometimes, it is convenient to use different directions for recursion and associativity. To build the right-recursive tree from the left recursive grammar, we could use a constructor that adds successive elements to the end of the list. A straightforward implementation of this idea would have to walk the list on each reduction, making the constructor itself take $\mathsf{O}(n^2)$ time, where n is the length of the list. Another classic solution to this problem uses a list-header node that contains pointers to both the first and last node in the list. This introduces an extra node to the list. If the system constructs many short lists, the overhead may be a problem.

A solution that we find particularly appealing is to use a list header node during construction, and to discard it when the list is fully built. Rewriting the grammar to use an ϵ-production makes this particularly clean:

Grammar			*Actions*
List	→	ϵ	`{$$ ← MakeListHeader();}`
	|	*List Elt*	`{$$ ← AddToEnd($1,$2);}`
Quux	→	*List*	`{$$ ← RemoveListHeader($1);}`

A reduction with the ϵ-production creates the temporary list header node; with a shift-reduce parser, this reduction occurs first. On a reduction for *List* → *List Elt*, the action invokes a constructor that relies on the presence of the temporary header node. When *List* is reduced on the right-hand side of any other production, the corresponding action invokes a function that discards the temporary header and returns the first element of the list. This solution lets the parser reverse the associativity at the cost of a small constant overhead in both space and time. It requires one more reduction per list, for the ϵ-production.

Thompson's Construction Syntax-directed actions can be used to perform more complex tasks. For example, Section 2.7.1 introduced Thompson's construction for building a non-deterministic finite automaton from a regular expression. Essentially, the construction has a small "pattern" NFA for each of the four base cases in a regular expression: (1) recognizing a symbol, (2) concatenating two REs, (3) alternation to choose one of two REs, and (4) closure to specify zero or more occurrences of an RE. The syntax of the regular expressions, themselves, might be specified with a grammar that resembles the expression grammar.

$$
\begin{array}{lcl}
\textit{RegExpr} & \rightarrow & \textit{RegExpr Alternation} \\
 & | & \textit{Alternation} \\
\textit{Alternation} & \rightarrow & \textit{Alternation}\ \underline{|}\ \textit{Closure} \\
 & | & \textit{Closure} \\
\textit{Closure} & \rightarrow & \textit{Closure}^{*} \\
 & | & \underline{(}\ \textit{RegExpr}\ \underline{)} \\
 & | & \texttt{symbol}
\end{array}
$$

The NFA construction algorithm fits naturally into a syntax-directed translation scheme based on this grammar. The actions follow Thompson's construction. On a case-by-case basis, it takes the following actions:

1. *single symbol:* On a reduction of *Closure* → `symbol`, the action constructs a new NFA with two states and a transition from the start state to the final state on the recognized symbol.

2. *closure:* On a reduction of *Closure* → *Closure**, the action adds an ϵ-move to the the NFA for closure, from each of its final states to its initial state.

```
RegExpr      →  RegExpr Alternation
                { add an $\epsilon$-move from the final state of $1
                  to start state of $2
                  $$ ← resulting NFA }
             |  Alternation
                { $$ ← $1 }
Alternation  →  Alternation | Closure
                { create new states $s_i$ and $s_j$
                  add an edge from $s_i$ to start states for $1 & $3
                  add an edge from each final state of $1 & $3 to $s_j$
                  designate $s_j$ as sole final state of the new NFA
                  $$ ← resulting NFA }
             |  Closure
                { $$ ← $1 }
Closure      →  Closure*
                { add an $\epsilon$-move from final states of
                  NFA for closure to its start state
                  $$ ← $1 }
             |  ( RegExpr )
                { $$ ← $2 }
             |  symbol
                { create two-state NFA for symbol
                  $$ ← the new NFA }
```

Figure 4.9: Syntax-directed version of Thompson's construction

3. *alternation:* On a reduction of *Alternation* → *Alternation* | *Closure*, the action creates new start and final states, s_i and s_j respectively. It adds an ϵ-move from s_i to the start state of the NFAs for alternation and closure. It adds an ϵ-move from each final state of the NFAs for alternation and closure to s_j. All states other than s_j are marked as non-final states.

4. *concatenation:* On a reduction of *RegExpr* → *RegExpr Alternation* the action adds an ϵ-move from each final state of the NFA for the first rexpr to the start state of the NFA for the second rexpr. It also changes the final states of the first rexpr into non-final states.

The remaining productions pass the NFA that results from higher-precedence sub-expressions upwards in the parse. In a shift-reduce parser, some representation of the NFA can be pushed onto the stack. Figure 4.9 shows how this would be accomplished in an *ad hoc* syntax-directed translation scheme.

Digression: What about Context-Sensitive Grammars?

Given the progression of ideas from the previous chapters, it might seem natural to consider the use of context-sensitive languages (CSLs) to address these issues. After all, we used regular languages to perform lexical analysis, and context-free languages to perform syntax analysis. A natural progression might suggest the study of CSLs and their grammars. Context-sensitive grammars (CSGs) can express a larger family of languages than can CFGs.

However, CSGs are not the right answer for two distinct reasons. First, the problem of parsing a CSG is P-space complete. Thus, a compiler that used CSG-based techniques could run quite slowly. Second, many of the important questions are difficult to encode in a CSG. For example, consider the issue of declaration before use. To write this rule into a CSG would require distinct productions for each combination of declared variables. With a sufficiently small name space, this might be manageable; in a modern language with a large name space, the set of names is too large to encode into a CSG.

4.5 What Questions Should the Compiler Ask?

Recall that the goal of context-sensitive analysis is to prepare the compiler for the optimization and code generation tasks that will follow. The questions that arise in context-sensitive analysis, then, divide into two broad categories: checking program legality at a deeper level than is possible with the CFG, and elaborating the compiler's knowledge base from non-local context to prepare for optimization and code generation.

Along those lines, many context-sensitive questions arise.

- Given a variable `a`, is it a scalar, an array, a structure, or a function? Is it declared? In which procedure is it declared? What is its datatype? Is it actually assigned a value before it is used?

- For an array reference `b[i,j,k]`, is `b` declared as an array? How many dimensions does `b` have? Are `i`, `j`, and `k` declared with a data type that is valid for an array index expression? Do `i`, `j`, and `k` have values that place `b[i,j,k]` inside the declared bounds of `b`?

- Where can `a` and `b` be stored? How long must their values be preserved? Can either be kept in a hardware register?

- In the reference `*c`, is `c` declared as a pointer? Is `*c` an object of the appropriate type? Can `*c` be reached via another name?

- How many arguments does the function `fee` take? Do all of its invocations pass the right number of arguments? Are those arguments of the correct type?

- Does the function `fie()` return a known constant value? Is it a pure function of its parameters, or does the result depend on some implicit state (*i.e.*, the values of static or global variables)?

Most of these questions share a common trait. Their answers involve information that is not locally available in the syntax. For example, checking the number and type of arguments at a procedure call requires knowledge of both the procedure's declaration and the call site in question. In many cases, these two statements will be separated by intervening context.

4.6 Summary and Perspective

In Chapters 2 and 3, we saw that much of the work in a compiler's front end can be automated. Regular expressions work well for lexical analysis. Context-free grammars work well for syntax analysis. In this chapter, we examined two ways of performing context-sensitive analysis.

The first technique, using attribute grammars, offers the hope of writing high-level specifications that produce reasonably efficient executables. Attribute grammars have found successful application in several domains, ranging from theorem provers through program analysis. (See Section 9.6 for an application in compilation where attribute grammars may be a better fit.) Unfortunately, the attribute grammar approach has enough practical problems that it has not been widely accepted as the paradigm of choice for context sensitive analysis.

The second technique, called *ad hoc* syntax-directed translation, integrates arbitrary snippets of code into the parser and lets the parser provide sequencing and communication mechanisms. This approach has been widely embraced, because of its flexibility and its inclusion in most parser generator systems.

Questions

1. Sometimes, the compiler writer can move an issue across the boundary between context-free and context-sensitive analysis. For example, we have discussed the classic ambiguity that arises between function invocation and array references in Fortran 77 (and other languages). These constructs might be added to the classic expression grammar using the productions:

$$\begin{array}{lcl} \textit{factor} & \rightarrow & \texttt{ident}\ \underline{(}\ \textit{ExprList}\ \underline{)} \\ \textit{ExprList} & \rightarrow & \textit{expr} \\ & | & \textit{ExprList}\ \underline{,}\ \textit{expr} \end{array}$$

   Unfortunately, the only difference between a function invocation and an array reference lies in how the `ident` is declared.

   In previous chapters, we have discussed using cooperation between the scanner and the parser to disambiguate these constructs. Can the problem be solved during context-sensitive analysis? Which solution is preferable?

2. Sometimes, a language specification uses context-sensitive mechanisms to check properties that can be tested in a context free way. Consider the grammar fragment in Figure 4.7. It allows an arbitrary number of *StorageClass* specifiers when, in fact, the standard restricts a declaration to a single *StorageClass* specifier.

(a) Rewrite the grammar to enforce the restriction grammatically.

(b) Similarly, the language allows only a limited set of combinations of *TypeSpecifier*. `long` is allowed with either `int` or `float`; `short` is allowed only with `int`. Either `signed` or `unsigned` can appear with any form of `int`. `signed` may also appear on `char`. Can these restrictions be written into the grammar?

(c) Propose an explanation for why the authors structured the grammar as they did. (Hint: the scanner returned a single token type for any of the *StorageClass* values and another token type for any of the *TypeSpecifiers*.)

(d) How does this strategy affect the speed of the parser? How does it change the space requirements of the parser?

3. Sometimes, a language design will include syntactic constraints that are better handled outside the formalism of a context-free grammar, even though the grammar can handle them. Consider, for example, the following "check off" keyword scheme:

$$\begin{array}{lcl@{\qquad}lcl}
\textit{phrase} & \rightarrow & \texttt{keyword}\ \alpha\ \beta\ \gamma\ \delta\ \zeta & \gamma & \rightarrow & \gamma\textit{-keyword} \\
 & & & & \mid & \epsilon \\
\alpha & \rightarrow & \alpha\textit{-keyword} & \delta & \rightarrow & \delta\textit{-keyword} \\
 & \mid & \epsilon & & \mid & \epsilon \\
\beta & \rightarrow & \beta\textit{-keyword} & \zeta & \rightarrow & \zeta\textit{-keyword} \\
 & \mid & \epsilon & & \mid & \epsilon
\end{array}$$

with the restrictions that α-keyword, β-keyword, γ-keyword, δ-keyword, and ζ-keyword appear in order, and that each of them appear at most once.

(a) Since the set of combinations is finite, it can clearly be encoded into a series of productions. Give one such grammar.

(b) Propose a mechanism using *ad hoc* syntax-directed translation to achieve the same result.

(c) A simpler encoding, however, can be done using a more permissive grammar and a hard-coded set of checks in the associated actions.

(d) Can you use an ϵ-production to further simply your syntax-directed translation scheme?

Chapter 6

Intermediate Representations

6.1 Introduction

In designing algorithms, a critical distinction arises between problems that must be solved *online*, and those that can be solved *offline*. In general, compilers work offline—that is, they can make more than a single pass over the code being translated. Making multiple passes over the code should improve the quality of code generated by the compiler. The compiler can gather information in one pass and use that information to make decisions in later passes.

The notion of a multi-pass compiler (see Figure 6.1) creates the need for an intermediate representation for the code being compiled. In translation, the compiler must derive facts that have no direct representation in the source code—for example, the addresses of variables and procedures. Thus, it must use some internal form—an intermediate representation or IR—to represent the code being analyzed and translated. Each pass, except the first, consumes IR. Each pass, except the last, produces IR. In this scheme, the intermediate representation becomes the definitive representation of the code. The IR must be expressive enough to record all of the useful facts that might be passed between phases of the compiler. In our terminology, the IR includes auxiliary tables, like a symbol table, a constant table, or a label table.

Selecting an appropriate IR for a compiler project requires an understanding of both the source language and the target machine, of the properties of programs that will be presented for compilation, and of the strengths and weaknesses of the language in which the compiler will be implemented.

Each style of IR has its own strengths and weaknesses. Designing an appropriate IR requires consideration of the compiler's task. Thus, a source-to-source translator might keep its internal information in a form quite close to the source; a translator that produced assembly code for a micro-controller might use an internal form close to the target machine's instruction set. It requires

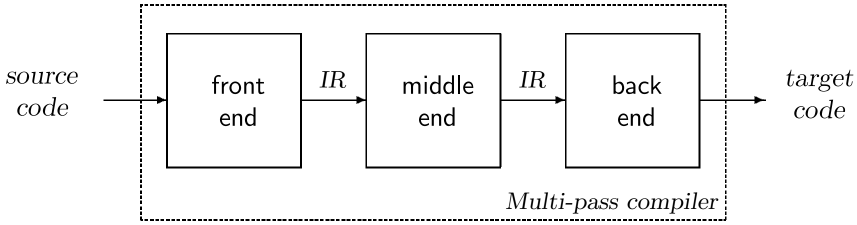


Figure 6.1: The role of IRs in a multi-pass compiler

consideration of the specific information that must be recorded, analyzed, and manipulated. Thus, a compiler for C might have additional information about pointer values that are unneeded in a compiler for PERL. It requires consideration of the operations that must be performed on the IR and their costs, of the range of constructs that must be expressed in the IR; and of the need for humans to examine the IR program directly.

(The compiler writer should never overlook this final point. A clean, readable external format for the IR pays for itself. Sometimes, syntax can be added to improve readability. An example is the ⇒ symbol used in the ILOC examples throughout this book. It serves no real syntactic purpose; however, it gives the reader direct help in separating operands from results.)

6.2 Taxonomy

To organize our thinking about IRs, we should recognize that there are two major axes along which we can place a specific design. First, the IR has a structural organization. Broadly speaking, three different organizations have been tried.

- Graphical IRs encode the compiler's knowledge in a graph. The algorithms are expressed in terms of nodes and edges, in terms of lists and trees. Examples include abstract syntax trees and control-flow graphs.

- Linear IRs resemble pseudo-code for some abstract machine. The algorithms iterate over simple, linear sequences of operations. Examples include bytecodes and three-address codes.

- Hybrid IRs combine elements of both structural and linear IRs, with the goal of capturing the strengths of both. A common hybrid representation uses a low-level linear code to represent blocks of straight-line code and a graph to represent the flow of control between those blocks.[1]

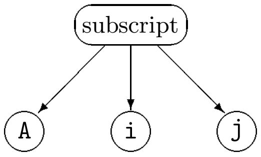
The structural organization of an IR has a strong impact on how the compiler writer thinks about analyzing, transforming, and translating the code. For example, tree-like IRs lead naturally to code generators that either perform a

[1] We say very little about hybrid IRs in the remainder of this chapter. Instead, we focus on the linear IRs and graphical IRs, leaving it to the reader to envision profitable combinations of the two.

tree-walk or use a tree pattern matching algorithm. Similarly, linear IRs lead naturally to code generators that make a linear pass over all the instructions (the "peephole" paradigm) or that use string pattern matching techniques.

The second axis of our IR taxonomy is the level of abstraction used to represent operations. This can range from a near-source representation where a procedure call is represented in a single node, to a low-level representation where multiple IR operations are assembled together to create a single instruction on the target machine.

To illustrate the possibilities, consider the difference between the way that a source-level abstract syntax tree and a low-level assembly-like notation might represent the reference `A[i,j]` into an array declared `A[1..10,1..10]`).



abstract syntax tree

```
load     1          ⇒ r1
sub      rj, r1     ⇒ r2
loadi    10         ⇒ r3
mul      r2, r3     ⇒ r4
sub      ri, r1     ⇒ r5
add      r4, r5     ⇒ r6
loadi    @A         ⇒ r7
loadA0   r7, r6     ⇒ rAij
```

low-level linear code

In the source-level AST, the compiler can easily recognize that the computation is an array reference; examining the low-level code, we find that simple fact fairly well obscured. In a compiler that tries to perform data-dependence analysis on array subscripts to determine when two different references can touch the same memory location, the higher level of abstraction in the AST may prove valuable. Discovering the array reference is more difficult in the low-level code; particularly if the IR has been subjected to optimizations that move the individual operations to other parts of the procedure or eliminate them altogether. On the other hand, if the compiler is trying to optimize the code generated for the array address calculation, the low-level code exposes operations that remain implicit in the AST. In this case, the lower level of abstraction may result in more efficient code for the address calculation.

The high level of abstraction is not an inherent property of tree-based IRs; it is implicit in the notion of a syntax tree. However, low-level expression trees have been used in many compilers to represent all the details of computations, such as the address calculation for `A[i,j]`. Similarly, linear IRs can have relatively high-level constructs. For example, many linear IRs have included a `mvcl` operation[2] to encode string-to-string copy as a single operation.

On some simple RISC machines, the best encoding of a string copy involves clearing out the entire register set and iterating through a tight loop that does a multi-word load followed by a multi-word store. Some preliminary logic is needed to deal with alignment and the special case of overlapping strings. By

[2] The acronym is for move character long, an instruction on the IBM 370 computers.

using a single IR instruction to represent this complex operation, the compiler writer can make it easier for the optimizer to move the copy out of a loop or to discover that the copy is redundant. In later stages of compilation, the single instruction is expanded, in place, into code that performs the copy or into a call to some system or library routine that performs the copy.

Other properties of the IR should concern the compiler writer. The costs of generating and manipulating the IR will directly effect the compiler's speed. The data space requirements of different IRs vary over a wide range; and, since the compiler typically touches all of the space that is allocated, data-space usually has a direct relationship to running time. Finally, the compiler writer should consider the expressiveness of the IR—its ability to accommodate all of the facts that the compiler needs to record. This can include the sequence of actions that define the procedure, along with the results of static analysis, profiles of previous executions, and information needed by the debugger. All should be expressed in a way that makes clear their relationship to specific points in the IR.

6.3 Graphical IRs

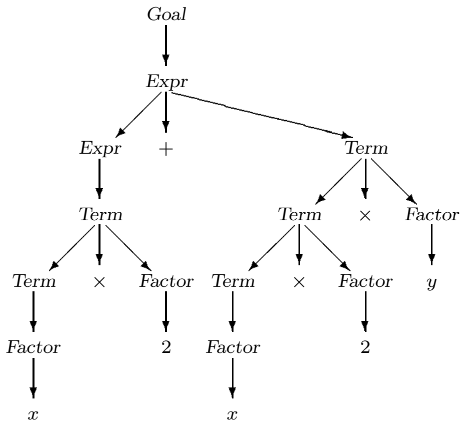
Many IRs represent the code being translated as a graph. Conceptually, all the graphical IRs consist of nodes and edges. The difference between them lies in the relationship between the graph and the source language program, and in the restrictions placed on the form of the graph.

6.3.1 Syntax Trees

The *syntax tree*, or *parse tree*, is a graphical representation for the derivation, or parse, that corresponds to the input program. The following simple expression grammar defines binary operations +, −, ×, and ÷ over the domain of tokens `number` and `id`.

$$
\begin{array}{lcl}
\mathit{Goal} & \rightarrow & \mathit{Expr} \\
\mathit{Expr} & \rightarrow & \mathit{Expr} + \mathit{Term} \\
 & | & \mathit{Expr} - \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \times \mathit{Factor} \\
 & | & \mathit{Term} \div \mathit{Factor} \\
 & | & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & \texttt{Number} \\
 & | & \texttt{Id}
\end{array}
$$

Simple Expression Grammar

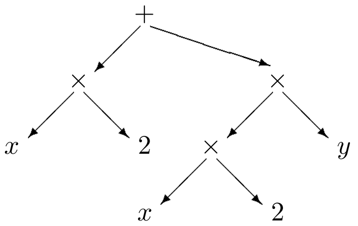


Syntax tree for $x \times 2 + x \times 2 \times y$

The syntax tree on the right shows the derivation that results from parsing the expression $x \times 2 + x \times 2 \times y$. This tree represents the complete derivation, with a node for each grammar symbol (terminal or non-terminal) in the derivation. It provides a graphic demonstration of the extra work that the parser goes through to maintain properties like precedence. Minor transformations on the grammar can reduce the number of non-trivial reductions and eliminate some of these steps. (See Section 3.6.2.) Because the compiler must allocate memory for the nodes and edges, and must traverse the entire tree several times, the compiler writer might want to avoid generating and preserving any nodes and edges that are not directly useful. This observation leads to a simplified syntax tree.

6.3.2 Abstract Syntax Tree

The *abstract syntax tree* (AST) retains the essential structure of the syntax tree, but eliminates the extraneous nodes. The precedence and meaning of the expression remain, but extraneous nodes have disappeared.

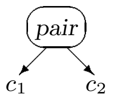


Abstract syntax tree for $x \times 2 + x \times 2 \times y$

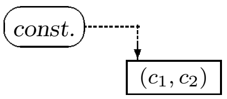
The AST is a near source-level representation. Because of its rough correspondence to the parse of the source text, it is easily built in the parser.

ASTs have been used in many practical compiler systems. Source-to-source systems, including programming environments and automatic parallelization tools, generally rely on an AST from which the source code can be easily regenerated. (This process is often called "pretty-printing;" it produces a clean source text by performing an inorder treewalk on the AST and printing each node as it is visited.) The S-expressions found in Lisp and Scheme implementations are, essentially, ASTs.

Even when the AST is used as a near-source level representation, the specific representations chosen and the abstractions used can be an issue. For example, in the $\mathcal{R}^n$ Programming Environment, the complex constants of Fortran programs, written $\texttt{(}c_1\texttt{,}c_2\texttt{)}$, were represented with the subtree on the left. This choice worked well for the syntax-directed editor, where the programmer was able to change c_1 and c_2 independently; the "*pair*" node supplied the parentheses and the comma.



AST *for editing*



AST *for compiling*

Digression: Storage Efficiency and Graphical Representations

Many practical systems have used abstract syntax trees to represent the source text being translated. A common problem encountered in these systems is the size of the AST relative to the input text. The AST in the $\mathcal{R}^n$ Programming Environment took up 1,000 bytes per Fortran source line—an amazing expansion. Other systems have had comparable expansion factors.

No single problem leads to this explosion in AST size. In some systems, it is the result of using a single size for all nodes. In others, it is the addition of myriad minor fields used by one pass or another in the compiler. Sometimes, the node size increases over time, as new features and passes are added.

Careful attention to the form and content of the AST can shrink its size. 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 less expensive to recompute information than to record it, write it, and read it.) In a few cases, we used hash-linking to record unusual facts—using one bit in the field that stores each node's type to indicate the presence of additional information stored in a hash table. (This avoided allocating fields that were rarely used.) To record the AST on disk, we went to a preorder treewalk; this eliminated any internal pointers.

In $\mathcal{R}^n$, the combination of all these things reduced to size of the AST in memory by roughly 75 percent. On disk, after the pointers were squeezed out, the files were about half that size.

However, this abstraction proved problematic for the compiler. Every part of the compiler that dealt with constants needed to include special case code to handle complex constants. The other constants all had a single `const` node that contained a pointer to a textual string recorded in a table. The compiler might have been better served by using that representation for the complex constant, as shown on the right. It would have simplified the compiler by eliminating much of the special case code.

6.3.3 Directed Acyclic Graph

One criticism of an AST is that it represents the original code too faithfully. In the ongoing example, $x \times 2 + x \times 2 \times y$, the AST contains multiple instances of the expression $x \times 2$. The *directed acyclic graph* (DAG) is a contraction of the AST that avoids unnecessary duplication. In a DAG, nodes can have multiple parents; thus, the two occurrences of $x \times 2$ in our example would be represented with a single subtree. Because the DAG avoids duplicating identical nodes and edges, it is more compact than the corresponding AST.

For expressions without assignment, textually identical expressions must produce identical values. The DAG for our example, shown in Figure 6.2, reflects this fact by containing only one copy of $x \times 2$. In this way, the DAG encodes an explicit hint about how to evaluate the expression. The compiler can generate code that evaluates the subtree for $x \times 2$ once and uses the result twice; know-

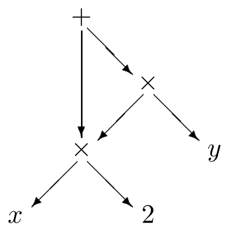


Figure 6.2: DAG for $x \times 2 + x \times 2 \times y$

ing that the subtrees are identical might let the compiler produce better code for the whole expression. The DAG exposed a redundancy in the source-code expression that can be eliminated by a careful translation.

The compiler can build a DAG in two distinct ways.

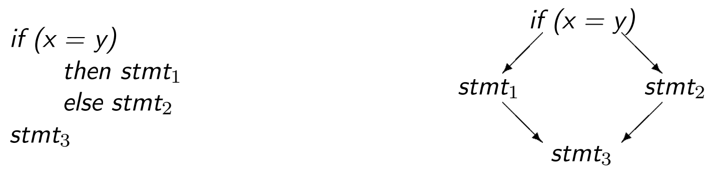
1. It can replace the constructors used to build an AST with versions that remember each node already constructed. To do this, the constructors would record the arguments and results of each call in a hash table. On each invocation, the constructor checks the hash table; if the entry already exists, it returns the previously constructed node. If the entry does not exist, it builds the node and creates a new entry so that any future invocations with the same arguments will find the previously computed answer in the table. (See the discussion of "memo functions" in Section 14.2.1.)

2. It can traverse the code in another representation (source or IR) and build the DAG. The DAG construction algorithm for expressions (without assignment) closely resembles the single-block value numbering algorithm (see Section 14.1.1). To include assignment, the algorithm must invalidate subtrees as the values of their operands change.

Some Lisp implementations achieve a similar effect for lists constructed with the `cons` function by using a technique called "hash-consing." Rather than simply constructing a `cons` node, the function consults a hash table to see if an identical node already exists. If it does, the `cons` function returns the value of the pre-existing `cons` node. This ensures that identical subtrees have precisely one representation. Using this technique to construct the AST for $x \times 2 + x \times 2 \times y$ would produce the DAG directly. Since hash-consing relies entirely on textual equivalence, the resulting DAG could not be interpreted as making any assertions about the value-equivalence of the shared subtrees.

6.3.4 Control-Flow Graph

The *control-flow graph* (CFG) models the way that the control transfers between the blocks in the procedure. The CFG has nodes that correspond to basic blocks in the procedure being compiled. Edges in the graph correspond to possible



```
if (x = y)
      then stmt1
      else stmt2
stmt3
```

A simple code fragment

Its control-flow graph

Figure 6.3: The control-flow graph

transfers of control between basic blocks. The CFG provides a clean graphical representation of the possibilities for run-time control flow. It is one of the oldest representations used in compilers; Lois Haibt built a CFG for the register allocator in the original IBM Fortran compiler.

Compilers typically use a CFG in conjunction with another IR. The CFG represents the relationships between blocks, while the operations inside a block are represented with another IR, such as an expression-level AST, a DAG, or a linear three-address code.

Some authors have advocated CFGs that use a node for each statement, at either the source or machine level. This formulation leads to cleaner, simpler algorithms for analysis and optimization; that improvement, however, comes at the cost of increased space. This is another engineering tradeoff; increasing the size of the IR simplifies the algorithms and increases the level of confidence in their correctness. If, in a specific application, the compiler writer can afford the additional space, a larger IR can be used. As program size grows, however, the space and time penalty can become significant issues.

6.3.5 Data-dependence Graph or Precedence Graph

Another graphical IR that directly encodes the flow of data between definition points and use points is the *dependence graph*, or *precedence graph*. Dependence graphs are an auxiliary intermediate representation; typically, the compiler constructs the DG to perform some specific analysis or transformation that requires the information.

To make this more concrete, Figure 6.4 reproduces the example from Section 1.3 on the left and shows its data-dependence graph is on the right. Nodes in the dependence graph represent definitions and uses. An edge connects two nodes if one uses the result of the other. A typical dependence graph does not model control flow or instruction sequencing, although the latter can be be inferred from the graph (see Chapter 11). In general, the data-dependence graph is not treated as the the compiler's definitive IR. Typically, the compiler maintains another representation, as well.

Dependence graphs have proved useful in program transformation. They are used in automatic detection of parallelism, in blocking transformations that improve memory access behavior, and in instruction scheduling. In more sophisticated applications of the data-dependence graph, the compiler may perform

```
1. | loadAI   r0, 0  ⇒ r1
2. | add      r1, r1 ⇒ r1
3. | loadAI   r0, 8  ⇒ r2
4. | mult     r1, r2 ⇒ r1
5. | loadAI   r0, 16 ⇒ r2
6. | mult     r1, r2 ⇒ r1
7. | loadAI   r0, 24 ⇒ r2
8. | mult     r1, r2 ⇒ r1
9. | storeAI  r1     ⇒ r0,0
```

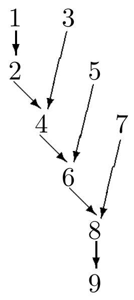


Figure 6.4: The data-dependence graph

extensive analysis of array subscript values to determine when references to the same array can overlap.

6.3.6 Static Single Assignment Graph

The *static single assignment graph* (SSA) directly represents the flow of values in a manner similar to the data-dependence graph. SSA form was designed to help in analyzing and improving code in a compiler. It lacks the clear relationship to original syntax found in a syntax tree or an AST. Similarly, it does not cleanly encode the flow of control; deriving control-flow relationships from the SSA graph may take some analysis, just as it would from the AST. However, its orientation toward analysis and optimization can pay off in an optimizing compiler (see Chapters 13 and 14).

In the SSA graph, the nodes are individual statements in the code. An edge runs from each reference to a value (a use) back to the node where the value was created (a definition). In a direct implementation of this notion, each reference has one or more edges that lead back to definitions. SSA simplifies this picture with the addition of two simple rules.

1. Each definition has a unique name.
2. Each use refers to a single definition.

These two constraints simplify both the SSA graph and any code that manipulates it. However, to reconcile these two rules with the reality of imperative programs, the compiler must modify the code. It must create a new name space and insert some novel instructions into the code to preserve its meaning under this new name space.

SSA Names To construct SSA form, the compiler must make a pass over the code to rename each definition and each use. To make the relationship between SSA-names and original names explicit, all the literature on SSA creates new names by adding subscripts to the original names. Thus, the first definition of x becomes x_0, while the second becomes x_1. Renaming can be done in a linear pass over the code.

Digression: Building SSA

Static single assignment form is the only IR we describe that does not have an obvious construction algorithm. While the efficient algorithm is complex enough to merit its own section in Chapter 13, a sketch of the construction process will clarify some of the mysteries surrounding SSA.

Assume that the input program is already in ILOC form. To convert it to an equivalent linear form of SSA, the compiler must:

1. insert ϕ-functions
2. rename ILOC-virtual registers

The difficult parts of the algorithm involve determining where ϕ-functions are needed and managing the renaming process to work efficiently. The simplest SSA-construction algorithm would insert a ϕ-function for each ILOC-virtual register at the start of each basic block that has more than one predecessor in the control-flow graph. (This inserts many unneeded ϕ-functions.)

To rename variables, the compiler can process the blocks, in a depth-first order. When a definition of $\mathtt{r}_i$ is encountered, it increments the current subscript for $\mathtt{r}_i$. At the end of a block, it looks down each control-flow edge and rewrites the appropriate ϕ-function parameter in each block that has multiple predecessors.

Of course, many bookkeeping details must be handled to make this work. The resulting SSA form has extraneous ϕ-functions. Some of these could be eliminated by noticing that a ϕ-function has identical arguments for each entering edge. For example, the operation $\mathtt{x}_{17} \leftarrow \phi(\mathtt{x}_7,\mathtt{x}_7,\mathtt{x}_7)$ serves no useful purpose. More precise algorithms for SSA construction eliminate most of the unneeded ϕ-functions.

ϕ-functions To reconcile the conflict that arises when distinct values flow along different paths into a single basic block, the SSA construction introduces new statements, called ϕ-functions. A ϕ-function takes as arguments the SSA-names for the value on each control-flow edge entering the block. It defines a new SSA name for use in subsequent references. When control enters a block, all of its ϕ-functions execute concurrently. Each one defines its output SSA-name with the value of its argument that corresponds to the most recently traversed control-flow edge.

These properties make SSA-form a powerful tool. The name space eliminates any issues related to the lifetime of a value. Since each value is defined in exactly one instruction, it is available along any path that proceeds from that instruction. The placement of ϕ-functions provides the compiler with information about the flow of values; it can use this information to improve the quality of code that it generates. The combination of these properties creates a representation that allows for accurate and efficient analysis.

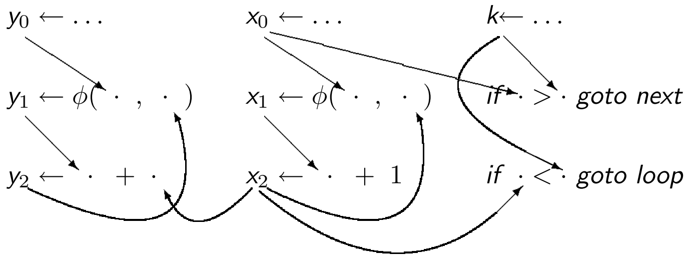
We have described SSA as a graphical form. Many implementations embed this graphical form into a linear IR by introducing a new IR operator for the

```
$x \leftarrow \cdots$
$y \leftarrow \cdots$
while($x < k$)
    $x \leftarrow x + 1$
    $y \leftarrow y + x$
```

Original code

```
       $x_0 \leftarrow \cdots$
       $y_0 \leftarrow \cdots$
       if ($x_0 > k$) goto next
loop:      $x_1 \leftarrow \phi(x_0,x_2)$
           $y_1 \leftarrow \phi(y_0,y_2)$
           $x_2 \leftarrow x_1 + 1$
           $y_2 \leftarrow y_1 + x_2$
           if ($x_2 < k$) goto loop
next:  $\cdots$
```

A linear SSA*-form*



$y_0 \leftarrow \ldots$
$x_0 \leftarrow \ldots$
$k \leftarrow \ldots$
$y_1 \leftarrow \phi(\cdot\ ,\ \cdot)$
$x_1 \leftarrow \phi(\cdot\ ,\ \cdot)$
if $\cdot > \cdot$ goto next
$y_2 \leftarrow \cdot + \cdot$
$x_2 \leftarrow \cdot + 1$
if $\cdot < \cdot$ goto loop

the SSA*-graph*

Figure 6.5: Static single assignment form

ϕ-function and directly renaming values in the code. This approach is also useful; the reader should be aware of both approaches to implementing SSA. The linear encoding of SSA can be used as a definitive IR, because the original linear IR encodes all the necessary information about control-flow that is missing or implicit in a pure SSA-graph. Figure 6.5 shows a small code fragment, along with its SSA-graph and the equivalent linear form.

The linear form displays 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 containing the ϕ-function. Thus, when the ϕ first executes, one of its arguments has never been defined. In many programming language contexts, this would cause problems. The careful definition of the ϕ-function's meaning avoids any problem. The ϕ behaves as if it were a copy from the argument corresponding to the entering edge into its output. Thus, along the edge that enters the loop, the ϕ selects x_0. Along the loop-closing edge, it selects x_2. Thus, it can never actually use the undefined variable.

The concurrent execution of ϕ-functions requires some care as well. Consider what would happen entering a block for the first time if it began with the following sequence of assignments:

$$\begin{aligned} x_1 &\leftarrow \phi(x_0, x_2) \\ y_1 &\leftarrow \phi(x_1, y_2) \end{aligned}$$

Since the two ϕ-functions are defined to execute concurrently, they read their arguments, then define their outputs. If the first entry was along the path corresponding to the second argument, the meaning is well defined. However, along the other path, y_1 receives the uninitialized versions of x_1, even though the clear intent is that it receive the value of x_1 after execution of the ϕ-function. While this example seems contrived, it is a simplified version of a problem that can arise if the compiler applies transformations to the SSA-form that rewrite names. In the translation from SSA-form back into executable code, the compiler must take care to recognize situations like this and generate code that serializes the assignments in the appropriate order.

6.4 Linear IRs

The alternative to structural IRs is, quite naturally, a linear form of IR. Some of the earliest compilers used linear forms; this was a natural notation for the authors, since they had previously programmed in assembly code.

The logic behind using linear forms is simple; the compiler must eventually emit a stream of instructions for the code being translated. That target machine code is almost always a linear form. Linear IRs impose a clear and useful ordering on the sequence of operations; for example, contrast the linear form of SSA with the graphical form, both shown in Figure 6.5.

When a linear form is used as the definitive representation, it must include a mechanism to encode transfers of control between different points in the program. This is usually done with branch and conditional branch operations. The code in a linear representation can be divided into basic blocks, or maximal length sequences of straight-line code, and the control-flow related operations that begin and end basic blocks. For our current purposes, every basic block begins with a label and ends with a branch. We include the branch at the end of the block, even if it is not strictly necessary. This makes it easier to manipulate the blocks; it eliminates implicit ordering between blocks that might otherwise exist.

6.4.1 One-Address Code

One-address code, also called stack machine code, assumes the presence of an operand stack. Most operations manipulate the stack; for example, an integer subtract operation would remove the top two elements from the stack and push their difference onto the stack (`push(pop() - pop())`, assuming left to right evaluation). The stack discipline creates a need for some new operations; for example, stack IRs usually include a `swap` operation that interchanges the top two elements of the stack. Several stack-based computers have been built; one-address code seems to have appeared in response to the demands of compiling for these machines. The left column of Figure 6.6 shows an example.

```
push      2
push      y
multiply
push      x
subtract
```

one-address code

```
loadi   2  ⇒ t1
load    y  ⇒ t2
mult    t2 ⇒t1
load    x  ⇒t3
sub     t1 ⇒t3
```

two-address code

```
loadi  2        ⇒ t1
load   y        ⇒ t2
mult   t1,t2    ⇒ t3
load   x        ⇒ t4
sub    t4, t3 ⇒ ⇒ t5
```

three-address code

Figure 6.6: Linear representations of $x - 2 \times y$

One-address code is compact. The stack creates an implicit name space that does not need to be represented in the IR. This shrinks the size of a program in IR form. Stack machine code is simple to generate and execute. Building complete programming language implementations on a stack IR requires the introduction of some control-flow constructs to handle branching and conditional evaluation. Several languages have been implemented in this way.

The compact nature of one-address code makes it an attractive way of encoding programs in environments where space is at a premium. It has been used to construct bytecode interpreters for languages like Smalltalk-80 and Java. It has been used as a distribution format for complex systems that must be transmitted in a compact form (over a relatively slow network, for example).

6.4.2 Two-Address Code

Two-address code is another linear IR. Two-address code allows statements of the form `x` ← `x` ⟨*op*⟩ `y`, with a single operator and, at most, two names. The middle column in Figure 6.6 shows an expression encoded in two-address form. Using two names saves space in the instruction format, at the cost of introducing destructive operations. The operation overwrites one of its operands; in practice, this can become a code shape problem.

For commutative operators, the compiler must decide which operand to preserve and which operand to destroy. With non-commutative operators, like shift, the choice is dictated by the definition of the IR. Some non-commutative operations are implemented in both ways—for example, a reverse subtract operator. When it chooses operands to preserve and to destroy, the compiler determines which values are available for subsequent use and reuse. Making these choices well is difficult.

Two-address code made technical sense when the compiler targeted a machine that used two-address, destructive operations (like the PDP-11). The destructive nature of these operations had a direct effect on the code that compilers could generate for these machines. Typical RISC machines offer the generality of three-address instructions; using a two-address code on these machines prematurely limits the space of values that can be named and used in the compiled code.

6.4.3 Three-Address Code

The term *three-address code* is used to describe many different representations. In practice, they all have a common feature; they admit statements of the form $x \langle op \rangle\ y \Rightarrow z$ with a single operator and, at most, three names. Practical forms of three-address code have operators that take fewer operands; load immediate is a common example. If SSA is being encoded into the three-address IR, the compiler writer may add a mechanism for representing arbitrary arity ϕ-functions. Three-address code can include forms of prefix or postfix code. The rightmost column of Figure 6.6 shows an expression translated into three-address code.

Three-address code is attractive for several reasons. The code is compact, relative to most graphical representations. It introduces a new set of names for values; a properly chosen name space can reveal opportunities for generating better code. It resembles many actual computers, particularly RISC microprocessors.

Within three-address codes, there exists a wide variation on the specific operators and their level of abstraction. Often, a single IR will contain low-level operations, such as branches, loads, and stores with no addressing modes, along with relatively high-level operations like `mvcl`, `copy`, `max`, or `min`. The critical distinction is that operations involving internal control flow are encoded in a single high-level operation. This allows the compiler to analyze, transform, and move them without respect to their internal control flow. At a later stage in compilation, the compiler expands these high-level operations into lower level operations for final optimization and code generation.

Quadruples Quadruples are a straight forward implementation of three-address code. A quadruple has four fields, one operator, two operands, or sources, and a destination operand. The set of quadruples that represents a program is held in a $k \times 4$ array of small integers. All names are explicit. This data structure is easy to manipulate.

The ILOC code used throughout this book is an example of quadruples. The example code from Figure 6.6 would be represented as follows:

loadI	2		t_1
load	y		t_2
mult	t_1	t_2	t_3
load	x		t_4
sub	t_4	t_3	t_5

The primary advantage of quadruples is simplicity; this leads to fast implementations, since most compilers generate good code for simple array accesses.

Triples Triples are a more compact form of three-address code. In a triples representation, the index of the operation in the linear array of instructions is used as an implicit name. This eliminates one fourth of the space required by quadruples. The strength of this notation is its introduction of a unique and implicit name space for values created by operations. Unfortunately, the name space is directly related to instruction placement, so that reordering instructions is difficult. (The compiler must update all references with the new location.) Here is the short example in a triples form:

(1)	loadi	2	
(2)	load	y	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Using the implicit name space has reduced the necessary storage.

Indirect Triples Indirect triples address the primary shortcoming of triples—the difficulty of reordering statements. In the indirect triple representation, operations are represented in a table of triples. Additionally, the compiler keeps a list of "statements" that specifies the order in which the triples occur. This makes the implicit names in the triple array permanent; to reorder operations, the compiler simply moves pointers in the statement list.

Statements				
(1)	(1)	loadi	2	
(2)	(2)	load	y	
(3)	(3)	mult	(1)	(2)
(4)	(4)	load	x	
(5)	(5)	sub	(4)	(3)

Indirect triples have two advantages over quadruples. First, reordering statements involves moving a single pointer; in the quadruple representation it involved moving all four fields. Second, if the same statement occurs multiple times, it can be represented once and referenced several places in the statement list. (The equivalent savings can be obtained in a DAG or by hash-consing).

Static Single Assignment Form In Section 6.3.6, we described SSA as a graphical IR. An alternate way to use SSA is to embed it into a linear IR, such as a three-address code. In this scheme, the compiler writer simply adds a ϕ instruction to the IR and has the compiler modify the code appropriately. This requires insertion of ϕ instructions at the appropriate control-flow merge-points and renaming variables and values to reflect the SSA name space.

The linear form of SSA is an attractive alternative to a graphical representation. It can be interpreted as the graph when desired; it can be treated as a linear IR when that approach is preferable. If the compiler maintains some correspondence between names and instructions in the IR, either through a naming discipline or a lookaside table, the linear SSA can be interpreted as providing pointers from references back to the definitions whose values they use. Such pointers are useful in optimization; they are called *use-definition chains.*

The primary complication introduced by adding ϕ-instructions to ILOC is the fact that the ϕ-instruction needs an arbitrary number of operands. Consider the end of a case statement. The control-flow graph has an edge flowing from each individual case into the block immediately following the case statement. Thus, the ϕ-functions in that block need an argument position for each individual case in the preceding case statement.

Choosing Between Them Quadruples are simple, but they require the most space. Triples achieve a space savings of twenty-five percent by making it expensive to reorder operations. Indirect triples make reordering better and have the possibility of saving space by eliminating duplication. The real cost of indirect triples lies in the extra memory operations required to reach each statement; as memory latencies rise, this tradeoff becomes less desirable. SSA can be implemented in any of these schemes.

6.5 Mapping Values to Names

The IRs described in Sections 6.3 and 6.4 represent the various operations that form the source program. The choice of specific operations to represent the code determines, to a large extent, how the compiler can translate and optimize the code. The mapping of values to names also has a strong influence on the code that the compiler can generate. If the name space hides the relationship between values, the compiler may be unable to rediscover those connections. Similarly, by implicitly encoding information about values in the name space, the compiler can expose selected facts to subsequent analysis and optimization.

6.5.1 Naming Temporary Values

In translating source code into an IR, the compiler must invent names for many of the intermediate stages in the compiler. We refer to the set of names used to express a computation as the *name space* of the computation. The choice of name spaces has a surprisingly strong impact on the behavior of the compiler. The strategy for mapping names onto values will determine, to a large extent, which computations can be analyzed and optimized.

Consider again the example of an array reference `A[i,j]` shown in Section 6.2. The two IR fragments represent the same computation. The low-level IR exposes more details to the compiler. These details can be inferred from the AST and code can be generated. In a straight forward translation from the AST, each reference to `A[i,j]` will produce the same code in the executable, independent of context.

With the low-level IR, each intermediate step has its own name. This exposes those results to analysis and to transformation. In practice, most of the improvement that compilers can achieve in optimization arises from capitalizing on context. As an alternative to the linear code, the compiler could use a lower-level AST that exposed the entire address computation. This would probably use more space, but it would allow the compiler to examine the component parts of the address computation.

Naming is a critical part of SSA-form. SSA imposes a strict discipline that generates names for every value computed by the code—program variable or

Digression: The Impact of Naming

In the late 1980s, we built an optimizing compiler for Fortran. We tried several different naming schemes in the front-end. The first version generated a new temporary for each computation by bumping a “next register” counter. This produced large name spaces, *i.e.*, 985 names for the 210 line Singular Value Decomposition (SVD) routine from Forsythe, Malcolm, and Moler’s book on numerical methods [32]. The large name space caused problems with both space and speed. The data-flow analyzer allocated several sets for each basic block. Each set was the size of the name space.

The second version used a simple allocate/free protocol to conserve names. The front end allocated temporaries on demand and freed them when the immediate uses were finished. This produced smaller name spaces, *i.e.*, SVD used 60 names. This sped up compilation. For example, it reduced the time to compute LIVE variables for SVD by sixty percent.

Unfortunately, associating multiple expressions with a single temporary name obscured the flow of data 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, determined by entering the operator and operands into a hash table. This ensured that each occurrence of $r_{17} + r_{21}$ targets the same register.
2. In $\langle \texttt{op} \rangle\ \texttt{r}_i,\ \texttt{r}_j \Rightarrow \texttt{r}_k$, k is chosen so that $i < k$ and $j < k$.
3. Move operations, $\texttt{r}_i \Rightarrow \texttt{r}_j$, are allowed to have $i > j$, unless j represents a scalar program variable.. The virtual register corresponding to such a variable is only set by move operations. Expressions evaluate into their “natural” register, then are moved to a variable.
4. Whenever a store to memory occurs, like `store` $r_i \Rightarrow r_j$, it is immediately followed by a copy from r_i into the variable’s virtual register.

This scheme space used about 90 names for SVD, but exposed all the optimizations found with the first name space. The compiler used these rules until we adopted SSA-form, with its own naming discipline.

transitory intermediate value. This uniform treatment exposes many details to the scrutiny of analysis and improvement. It encodes information about where the value was generated. It provides a “permanent” name for the value, even if the corresponding program variable gets redefined. This can improve the results of analysis and optimization.

6.5.2 Memory Models

Just as the mechanism for naming temporary values affects the information that can be represented in an IR version of a program, so, too, does the compiler’s method of choosing storage locations for each value. The compiler must determine, for each value computed in the code, where that value will reside. For the code to execute, the compiler must assign a specific location such as register $\mathtt{r}_{13}$ or sixteen bytes from the label L0089. Before the final stages of code generation, however, the compiler may use symbolic addresses that encode a level in the memory hierarchy, *i.e.* registers or memory, but not a specific location within that level.

As an example, consider the ILOC examples used throughout the book. A symbolic memory address is denoted by prefixing it with the character @. Thus, @x is the offset of x from the start of its storage area. Since $\mathtt{r}_0$ holds the activation record pointer, an operation that uses @x and $\mathtt{r}_0$ to compute an address depends, implicitly, on the decision to store the variable x in the memory reserved for the current procedure's activation record.

In general, compilers work from one of two memory models.

Register-to-register model: Under this model, the compiler keeps values in registers aggressively, ignoring any limitations imposed by the size of the machine's physical register set. Any value that can legally be kept in a register is kept in a register. Values are stored to memory only when the semantics of the program require it—for example, at a procedure call, any local variable whose address is passed as a parameter to the called procedure must be stored back to memory. A value that cannot be kept in a register is stored in memory. The compiler generates code to store its value each time it is computed and to load its value at each use.

Memory-to-memory model: Under this model, the compiler assumes that all values are kept in memory locations. Values move from memory to a register just before they are used. Values move from a register to memory just after they are defined. The number of register names used in the IR version of the code is small compared to the register-to-register model. In a memory-to-memory model, the designer may find it worthwhile to include memory-to-memory forms of the basic operations, such as a memory-to-memory add.

The choice of memory model is mostly orthogonal to the choice of IR. The compiler writer can build a memory-to-memory AST or a memory-to-memory version of ILOC just easily as register-to-register versions of these IRs. (One-address code might be an exception; it contains its own unique memory model—the stack. A one-address format makes much less sense without the implicit naming scheme of stack-based computation.)

The choice of memory model has an impact on the rest of the compiler. For example, with a register-to-register model, the compiler must perform register allocation as part of preparing the code for execution. The allocator must map the set of virtual registers onto the physical registers; this often requires insertion of extra load, store, and copy operations. Under a register-to-register model, the allocator adds instructions to make the program behave correctly on the target machine. Under a memory-to-memory model, however, the pre-allocation code

Digression: The Hierarchy of Memory Operations in Iloc 9x

The ILOC used in this book is abstracted from an IR named ILOC 9X that was used in a research compiler project at Rice University. ILOC 9X includes a hierarchy of memory operations that the compiler uses to encode knowledge about values. These operations are:

Operation	*Meaning*
immediate load	loads a known constant value into a register
constant load	loads a value that does not change during execution. The compiler does not know the value, but can prove that it is not defined by a program operation.
scalar load & store	operates on a scalar value, not an array element, a structure element, or a pointer-based value.
general load & store	operates on a value that may be an array element, a structure element, or a pointer-based value. This is the general-case operation.

By using this hierarchy, the front-end can encode knowledge about the target value directly into the ILOC 9X code. As other passes discover additional information, they can rewrite operations to change a value from a general purpose load to a more restricted form. Thus, constant propagation might replace a general load or a scalar load with an immediate load. If an analysis of definitions and uses discovers that some location cannot be defined by any executable store operation, it can be rewritten as a constant load.

typically uses fewer registers than a modern processor provides. Thus, register allocation looks for memory-based values that it can hold in registers for longer periods of time. In this model, the allocator makes the code faster and smaller by removing some of the loads and stores.

Compilers for RISC machines tend to use the register-to-register model for two different reasons. First, the register-to-register model more closely reflects the programming model of RISC architecture. RISC machines rarely have a full complement of memory-to-memory operations; instead, they implicitly assume that values can be kept in registers. Second, the register-to-register model allows the compiler to encode subtle facts that it derives directly in the IR. The fact that a value is kept in a register means that the compiler, at some earlier point, had proof that keeping it in a register is safe.[3]

[3] If the compiler can prove that only one name provides access to a value, it can keep that value in a register. If multiple names might exist, it must behave conservatively and keep the

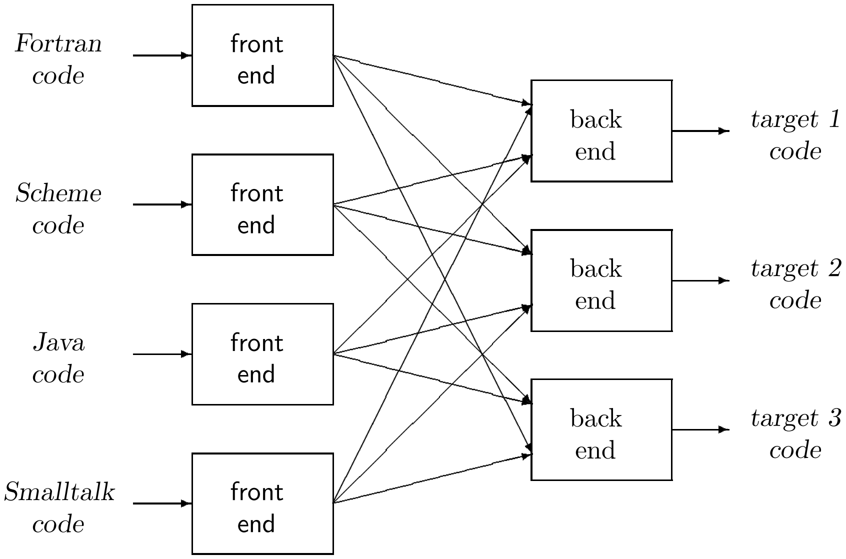


Figure 6.7: Developing a universal intermediate form

6.6 Universal Intermediate Forms

Because compilers are complex, programmers (and managers) regularly seek ways to decrease the cost of building them. A common, but profound, question arises in these discussions: can we design a "universal" intermediate form that will represent any programming language targeted for any machine? In one sense, the answer must be "yes"; we can clearly design an IR that is Turing-equivalent. In a more practical sense, however, this approach has not, in general, worked out well.

Many compilers have evolved from a single front end to having front ends for two or three languages that use a common IR and a single back end. This requires, of course, that the IR and the back end are both versatile enough to handle all the features of the different source languages. As long as they are reasonably similar languages, such as FORTRAN and C, this approach can be made to work.

The other natural direction to explore is retargeting the back end to multiple machines. Again, for a reasonably small collection of machines, this has been shown to work in practice.

Both of these approaches have had limited success in both commercial and research compilers. Taken together, however, they can easily lead to a belief that we can produce n front ends, m back ends, and end up with $n \times m$ compilers. Figure 6.7 depicts the result of following this line of thought too far. Several

value in memory. For example, a local variable `x` can be kept in a register, unless the program takes its address (`&x` in C) or passes it as a call-by-reference parameter to another procedure.

projects have tried to produce compilers for dissimilar languages and multiple machines, with the notion that a single IR can bind together all of the various components.

The more ambitious of these projects have foundered on the complexity of the IR. For this idea to succeed, the separation between front end, back end, and IR must be complete. The front end must encode all language-related knowledge and must encode no machine-specific knowledge. The back end must handle all machine-specific issues and have no knowledge about the source language. The IR must encode all the facts passed between front end and back end, and must represent all the features of all the languages in an appropriate way. In practice, machine-specific issues arise in front ends; language-specific issues find their way into back ends; and "universal" IRs become too complex to manipulate and use.

Some systems have been built using this model; the successful ones seem to be characterized by

- IRs that are near the abstraction level of the target machine
- target machines that are reasonably similar
- languages that have a large core of common features

Under these conditions, the problems in the front end, back end, and IR remain manageable. Several commercial compiler systems fit this description; they compile languages such as Fortran, C, and C++ to a set of similar architectures.

6.7 Symbol Tables

As part of translation, the compiler derives information about the various entities that the program manipulates. It must discover and store many distinct kinds of information. It will encounter a variety of names that must be recorded—names for variables, defined constants, procedures, functions, labels, structures, files, and computer-generated temporaries. For a given textual name, it might need a data type, the name and lexical level of its declaring procedure, its storage class, and a base address and offset in memory. If the object is an aggregate, the compiler needs to record the number of dimensions and the upper and lower bounds for each dimension. For records or structures, the compiler needs a list of the fields, along with the relevant information on each field. For functions and procedures, the compiler might need to know the number of parameters and their types, as well as any returned values; a more sophisticated translation might record information about the modification and use of parameters and externally visible variables.

Either the IR must store all this information, or the compiler must re-derive it on demand. For the sake of efficiency, most compilers record facts rather than recompute them. (The one common exception to this rule occurs when the IR is written to external storage. Such I/O activity is expensive relative to computation, and the compiler makes a complete pass over the IR when it reads the information. Thus, it can be cheaper to recompute information than to write it to external media and read it back.) These facts can be recorded directly in the IR. For example, a compiler that builds an AST might record information about variables as annotations (or attributes) on the nodes representing each variable's declaration. The advantage of this approach is that it uses a single representation for the code being compiled. It provides a uniform access method and a single implementation. The disadvantage of this approach is that the single access method may be inefficient—navigating the AST to find the appropriate declaration has its own costs. To eliminate this inefficiency, the compiler can thread the IR so that each reference has a link back to its declaration. This adds space to the IR and overhead to the IR-builder. (The next step is to use a hash table to hold the declaration link for each variable during IR construction—in effect, creating a symbol table.)

The alternative, as we saw in Chapter 4, is to create a central repository for these facts and to provide efficient access to it. This central repository, called a *symbol table*, becomes an integral part of the compiler's IR. The symbol table localizes information derived from distant parts of the source code; it simplifies the design and implementation of any code that must refer to information derived earlier in compilation. It avoids the expense of searching the IR to find the portion that represents a variable's declaration; using a symbol table often eliminates the need to represent the declarations directly in the IR. (An exception occurs in source-to-source translation. The compiler may build a symbol table for efficiency and preserve the declaration syntax in the IR so that it can produce an output program that closely resembles the input program.) It eliminates the overhead of making each reference contain a pointer back to the declaration. It replaces both of these with a computed mapping from the textual name back to the stored information. Thus, in some sense, the symbol table is simply an efficiency hack.

Throughout this text, we refer to "the symbol table." In fact, the compiler may include several distinct, specialized symbol tables. These include variable tables, label tables, tables of constants, and reserved keyword tables. A careful implementation might use the same access methods for all these tables. (The compiler might also use a hash table as an efficient representation for some of the sparse graphs built in code generation and optimization.)

Symbol table implementation requires attention to detail. Because nearly every aspect of translation refers back to the symbol table, efficiency of access is critical. Because the compiler cannot predict, before translation, the number of names that it will encounter, expanding the symbol table must be both graceful and efficient. This section provides a high-level treatment of the issues that arise in designing a symbol table. It presents the compiler-specific aspects of symbol table design and use. For deeper implementation details and design alternatives, the reader is referred to Section B.4.

6.7.1 Hash Tables

In implementing a symbol table, the compiler writer must choose a strategy for organizing and searching the table. Myriad schemes for organizing lookup tables exist; we will focus on tables indexed with a "hash function." Hashing,

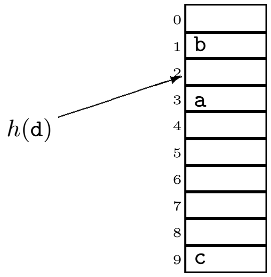


Figure 6.8: Hash-table implementation — the concept

as this technique is called, has an expected-case O(1) cost for both insertion and lookup. With careful engineering, the implementor can make the cost of expanding the table and of preserving it on external media quite reasonable. For the purposes of this chapter, we assume that the symbol table is organized as a simple hash table. Implementation techniques for hash tables have been widely studied and taught.

Hash tables are conceptually elegant. They use a *hash function*, h, to map names into small integers, and take the small integer as an index into the table. With a hashed symbol table, the compiler stores all the information that it derives about the name n in the table at $h(n)$. Figure 6.8 shows a simple ten-slot hash table. It is a vector of records, each record holding the compiler-generated description of a single name. The names `a`, `b`, and `c` have already been inserted. The name `d` is being inserted, at $h(\texttt{d}) = 2$.

The primary reason for using hash tables is to provide a constant-time lookup, keyed by a textual name. To achieve this, h must be inexpensive to compute, and it must produce a unique small integer for each name. Given an appropriate function h, accessing the record for n requires computing $h(n)$ and indexing into the table at $h(n)$. If h maps two or more symbols to the same small integer, a "collision" occurs. (In Figure 6.8, this would occur if $h(d) = 3$.) The implementation must handle this situation gracefully, preserving both the information and the lookup time. In this section, we assume that h is a perfect hash function—that is, it never produces a collision. Furthermore, we assume that the compiler knows, in advance, how large to make the table. Section B.4 describes hash-table implementation in more detail, including hash functions, collision handling, and schemes for expanding a hash table.

6.7.2 Building a Symbol Table

The symbol table defines two interface routines for the rest of the compiler.

`LookUp(`***name***`)` returns the record stored in the table at $h(name)$ if one exists. Otherwise, it returns a value indicating that *name* was not found.

`Insert(`*name,record*`)` stores the information in *record* in the table at $h(name)$. It may expand the table to accommodate the record for *name*.

Digression: An Alternative to Hashing

Hashing is the most widely used method for organizing a compiler's symbol table. Multiset discrimination is an interesting alternative that eliminates any possibility of worst-case behavior. The critical insight behind this technique is that the index can be constructed off-line in the scanner.

To use multiset discrimination for the symbol table, the compiler writer must take a different approach to scanning. Instead of processing the input incrementally, the compiler scans the entire program to find the complete set of identifiers. As it discovers each identifier, it creates a tuple $\langle$*name,position*$\rangle$, where *name* is the text of the identifier and *position* is its ordinal position in the list of all tokens. It enters all the tuples into a large multiset.

The next step lexicographically sorts the multiset. In effect, this creates a set of bags, one per identifier. Each bag holds the tuples for all of the occurrences of its identifier. Since each tuple refers back to a specific token, through its *position* value, the compiler can use the sorted multiset to rewrite the token stream. It makes a linear scan over the multiset, processing each bag in order. The compiler allocates a symbol table index for the entire bag, then rewrites the tokens to include that index. This augments the identifier tokens with their symbol table index. If the compiler needs a textual lookup function, the resulting table is ordered alphabetically for a binary search.

The price for using this technique is an extra pass over the token stream, along with the cost of the lexicographic sort. The advantages, from a complexity perspective, are that it avoids any possibility of hashing's worst case behavior, and that it makes the initial size of the symbol table obvious, even before parsing. This same technique can be used to replace a hash table in almost any application where an off-line solution will work.

The compiler needs separate functions for `LookUp` and `Insert`. (The alternative would have `LookUp` insert the name when it fails to find it in the table.) This ensures, for example, that a `LookUp` of an undeclared variable will fail—a property useful for detecting a violation of the declare-before-use rule in syntax-directed translation schemes, or for supporting nested lexical scopes.

This simple interface fits directly into the *ad hoc* syntax-directed translation scheme for building a symbol table, sketched in Section 4.4.3. In processing declaration syntax, the compiler builds up a set of attributes for the variable. When the parser reduces by a production that has a specific variable name, it can enter the name and attributes into the symbol table using `Insert`. If a variable name can appear in only one declaration, the parser can call `LookUp` first to detect a repeated use of the name. When the parser encounters a variable name outside the declaration syntax, it uses `LookUp` to obtain the appropriate information from the symbol table. `LookUp` fails on any undeclared name. The compiler writer, of course, may need to add functions to initialize the table, to store it and retrieve it using external media, and to finalize it. For a language with a single name space, this interface suffices.

6.7.3 Handling Nested Lexical Scopes

Few, if any, programming languages provide a single name space. Typically, the programmer manages multiple names spaces. Often, some of these name spaces are nested inside one another. For example, a C programmer has four distinct kinds of name space.

1. A name can have global scope. Any global name is visible in any procedure where it is declared. All declarations of the same global name refer to a single instance of the variable in storage.

2. A name can have a file-wide scope. Such a name is declared using the `static` attribute outside of a procedure body. A static variable is visible to every procedure in the file containing the declaration. If the name is declared static in multiple files, those distinct declarations create distinct run-time instances.

3. A name can be declared locally within a procedure. The scope of the name is the procedure itself. It cannot be referenced by name outside the declaring procedure. (Of course, the declaring procedure can take its address and store it where other procedures can reference the address. This may produce wildly unpredictable results if the procedure has completed execution and freed its local storage.)

4. A name can be declared within a block, denoted by a pair of curly braces. While this feature is not often used by programmers, it is widely used by macros to declare a temporary location. A variable declared in this way is only visible inside the declaring block.

Each distinct name space is called a *scope*. Language definitions includes rules that govern the creation and accessibility of scopes. Many programming languages include some form of nested lexical scopes.

Figure 6.9 shows some fragments of C code that demonstrate its various scopes. The level zero scope contains names declared as global or file-wide static. Both `example` and `x` are global, while `w` is a static variable with file-wide scope. Procedure `example` creates its own local scope, at level one. The scope contains `a` and `b`, the procedure's two formal parameters, and its local variable `c`. Inside `example`, curly braces create two distinct level two scopes, denoted as level *2a* and level *2b*. Level *2a* declares two variables, `b` and `z`. This new incarnation of `b` overrides the formal parameter `b` declared in the level one scope by `example`. Any reference to `b` inside the block that created *2a* names the local variable rather than the parameter at level one. Level *2b* declares two variables, `a` and `x`. Each overrides a variable declared in an outer scope. Inside level *2b*, another block creates level three and declares `c` and `x`.

All of this context goes into creating the name space in which the assignment statement executes. Inside level three, the following names are visible: `a` from *2b*, `b` from one, `c` from three, `example` from zero, `w` from zero, and `x` from three. No incarnation of the name `z` is active, since *2a* ends before three begins. Since

```
static int w;      /* level 0 */
int x;

void example(a,b);
  int a, b;        /* level 1 */
{
  int c;
  {
    int b, z;      /* level 2a */
      ...
  }
  {
    int a, x;      /* level 2b */
      ...
      {
        int c, x; /* level 3 */
        b = a + b + c + x;
      }
  }
}
```

Level	*Names*
0	`w, x, example`
1	`a, b, c`
2a	`b, z`
2b	`a, x`
3	`c, x`

Figure 6.9: Lexical scoping example

`example` at level zero is visible inside level three, a recursive call on `example` can be made. Adding the declaration "`int example`" to level *2b* or level three would hide the procedure's name from level three and prevent such a call.

To compile a program that contains nested scopes, the compiler must map each variable reference back to a specific declaration. In the example, it must distinguish between the multiple definitions of `a`, `b`, `c`, and `x` to select the relevant declarations for the assignment statement. To accomplish this, it needs a symbol table structure that can resolve a reference to the lexically most recent definition. At compile-time, it must perform the analysis and emit the code necessary to ensure addressability for all variables referenced in the current procedure. At run-time, the compiled code needs a scheme to find the appropriate incarnation of each variable. The run-time techniques required to establish addressability for variables are described in Chapter 8.

The remainder of this section describes the extensions necessary to let the compiler convert a name like x to a *static distance coordinate*—a ⟨*level*,*offset*⟩ pair, where *level* is the lexical level at which x's declaration appears and *offset* is an integer address that uniquely identifies the storage set aside for x. These same techniques can also be useful in code optimization. For example, the DVNT algorithm for discovering and removing redundant computations relies on a scoped hash table to achieve efficiency on extended basic blocks (see Section 12.1).

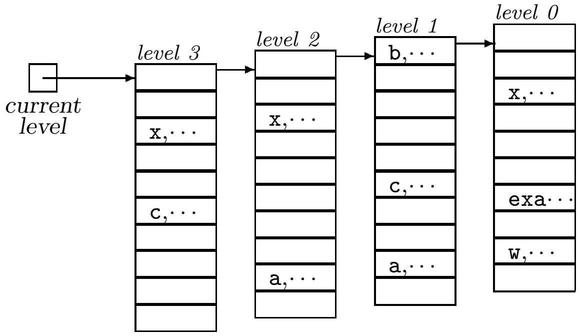


Figure 6.10: Simple "sheaf-of-tables" implementation

The Concept To manage nested scopes, the parser must change, slightly, its approach to symbol table management. Each time the parser enters a new lexical scope, it can create a new symbol table for that scope. As it encounters declarations in the scope, it enters the information into the current table. `Insert` operates on the current symbol table. When it encounters a variable reference, `LookUp` must first search the table for the current scope. If the current table does not hold a declaration for the name, it checks the table for the surrounding scope. By working its way through the symbol tables for successively lower lexical levels, it will either find the most recent declaration for the name, or fail in the outermost scope—indicating that the variable has no declaration visible from the current scope.

Figure 6.10 shows the symbol table built in this fashion for our example program, at the point where the parser has reached the assignment statement. When the compiler invokes the modified `LookUp` function for the name `b`, it will fail in level three, fail in level two, and find the name in level one. This corresponds exactly to our understanding of the program—the most recent declaration for `b` is as a parameter to `example`, in level one. Since the first block at level two, block *2a*, has already closed, its symbol table is not on the search chain. The level where the symbol is found, two in this case, forms the first part of the static distance coordinate for `b`. If the symbol table record includes a storage offset for each variable, then `LookUp` can easily return the static distance coordinate.

The Details To handle this scheme, two additional calls are required. The compiler needs a call that can initialize a new symbol table and one that can finalize the table for a level.

`InitializeScope()` increments the current level and creates a new symbol table for that level. It links the previous level's symbol table to the new table, and updates the current level pointer used by both `LookUp` and `Insert`.

`FinalizeScope()` changes the current level pointer so that it points at the

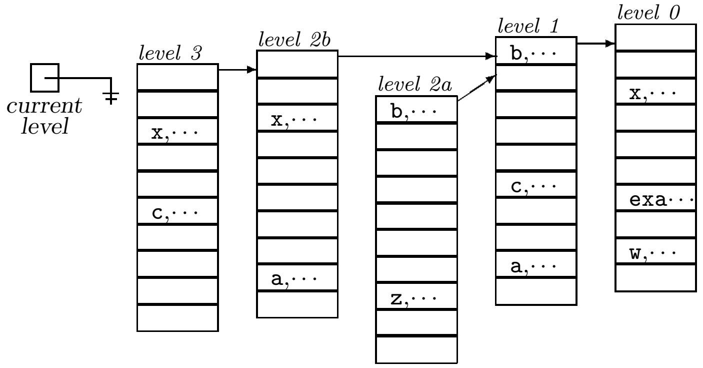


Figure 6.11: Final table for the example

table for the scope surrounding the current level, and then decrements the current level. If compiler needs to preserve the level-by-level tables for later use, `FinalizeLevel` can either leave the table intact in memory, or it can write the table to external media and reclaim its space. (In a system with garbage collection, `FinalizeLevel` should add the finalized table to a list of such tables.)

To account for lexical scoping, the parser should call `InitializeScope` each time it enters a new lexical scope and `FinalizeScope` each time it exits a lexical scope.

With this interface, the program in Figure 6.9 would produce the following sequence of calls

1. `InitializeScope`
2. `Insert(a)`
3. `Insert(b)`
4. `Insert(c)`
5. `InitializeScope`
6. `Insert(b)`
7. `Insert(z)`
8. `FinalizeScope`
9. `InitializeScope`
10. `Insert(a)`
11. `Insert(x)`
12. `InitializeScope`
13. `Insert(c)`
14. `Insert(x)`
15. `LookUp(b)`
16. `LookUp(a)`
17. `LookUp(b)`
18. `LookUp(c)`
19. `LookUp(x)`
20. `FinalizeScope`
21. `FinalizeScope`
22. `FinalizeScope`

As it enters each scope, the compiler calls `InitializeScope()`. It adds each variable to the table using `Insert()`. When it leaves a given scope, it calls `FinalizeScope()` to discard the declarations for that scope. For the assignment statement, it looks up each of the names, as encountered. (The order of the `LookUp()` calls will vary, depending on how the assignment statement is traversed.)

If `FinalizeScope` retains the symbol tables for finalized levels in memory, the net result of these calls will be the symbol table shown in Figure 6.11. The current level pointer is set to an invalid value. The tables for all levels are left in memory and linked together to reflect lexical nesting. The compiler can provide subsequent passes of the compiler with access to the relevant symbol table information by storing a pointer to the appropriate table into the IR at the start of each new level.

6.7.4 Symbol Table Contents

So far, the discussion has focused on the organization and use of the symbol table, largely ignoring the details of what information should be recorded in the symbol table. The symbol table will include an entry for each declared variable and each procedure. The parser will create these entries. As translation proceeds, the compiler may need to create additional variables to hold values not named explicitly by the source code. For example, converting $x - 2 \times y$ into ILOC creates a temporary name for the value $2 \times y$, and, perhaps, another for $x - 2 \times y$. Often, the compiler will synthesize a name, such as `t00017`, for each temporary value that it creates. If the compiler names these values and creates symbol table records for them, the rest of the compiler can treat them in the same way that it handles programmer-declared names. This avoids special-case treatment for compiler-generated variables and simplifies the implementation. Other items that may end up in the symbol table, or in a specialized auxiliary table, include literal constants, literal strings, and source code labels.

For each entry in the symbol table, the compiler will keep some set of information that may include: its textual name, its source-language data type, its dimensions (if any), the name and level of the procedure that contains its declaration, its storage class (global, static, local, *etc.*), and its offset in storage from the start of its storage class. For global variables, call-by-reference parameters, and names referenced through a pointer, the table may contain information about possible aliases. For aggregates, such as structures in C or records in Pascal, the table should contain an index into a table of structure information. For procedures and functions, the table should contain information about the number and type of arguments that it expects.

6.8 Summary and Perspective

The choice of intermediate representations has a major impact on the design, implementation, speed, and effectiveness of a compiler. None of the intermediate forms described in this chapter is, definitively, the right answer for all compilers. The designer must consider the overall goals of the compiler project when selecting an intermediate form, designing its implementation, and adding auxiliary data structures such as symbol and label tables.

Contemporary compiler systems use all manner of intermediate representations, 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 the Gnu compiler systems). Many compilers use multiple IRs—building a second or third IR to perform a particular analysis or transformation, then modifying the original, and definitive, IR to reflect the result.

Questions

1. In general, the compiler cannot pay attention to issues that are not represented in the IR form of the code being compiled. For example, performing register allocation on one-address code is an oxymoron.

   For each of the following representations, consider what aspects of program behavior and meaning are explicit and what aspects are implicit.

   (a) abstract syntax tree

   (b) static single assignment form

   (c) one-address code

   (d) two-address code

   (e) three-address code

   Show how the expression `x - 2` × `y` might be translated into each form.

   Show how the code fragment

   ```
   if (c[i] ≠ 0)
       then a[i] ← b[i] ÷ c[i];
       else a[i] ← b[i];
   ```

   might be represented in an abstract syntax tree and in a control-flow graph. Discuss the advantages of each representation. For what applications would one representation be preferable to the other?

2. Some part of the compiler must be responsible for entering each identifier into the symbol table. Should it be the scanner or the parser? Each has an opportunity to do so. Is there an interaction between this issue, declare-before-use rules, and disambiguation of subscripts from function calls in a language with the Fortran 77 ambiguity?

3. The compiler must store information in the IR version of the program that allows it to get back to the symbol table entry for each name. Among the options open to the compiler writer are pointers to the original character strings and subscripts into the symbol table. Of course, the clever implementor may discover other options.

   What are the advantages and disadvantages of each of these representations for a name? How would you represent the name?

   Symbol Tables: You are writing a compiler for your favorite lexically-scoped language.

   Consider the following example program:

```
              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;
```

(a) Draw the symbol table and its contents at the point labelled *here*.

(b) What actions are required for symbol table management when the parser enters a new procedure and when it exits a procedure?

Chapter Notes

The literature on intermediate representations and experience with them is sparse. This is somewhat surprising because of the major impact that decisions about IRs have on the structure and behavior of a compiler. The classic forms are described in textbooks dating back to the early 1970s. Newer forms like SSA are described in the literature as tools for analysis and transformation of programs.

In practice, the design and implementation of an IR has an inordinately large impact on the eventual characteristics of the completed compiler. Large, balky IRs seem to shape systems in their own image. For example, the large ASTs used in early 1980s programming environments like $\mathcal{R}^n$ limited the size of programs that could be analyzed. The RTL form used in LCC is rather low-level in its abstraction. Accordingly, the compiler does a fine job of managing details like those needed for code generation, but has few, if any, transformations that require source-like knowledge, such as loop blocking to improve memory hierarchy behavior.

Chapter 7

The Procedure Abstraction

7.1 Introduction

In an Algol-like language, the procedure is a programming construct that creates a clean, controlled, protected execution environment. Each procedure has its own private, named storage. Statements executed inside the procedure can access these private, or local, variables. Procedures execute when they are invoked, or called, by another procedure. The called procedure can return a value to its caller, in which case the procedure is termed a *function.* This 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 are the base unit of work for most compilers. Few systems require that the entire program be presented for compilation at one time. Instead, the compiler can process arbitrary collections of procedures. This feature, known as *separate compilation*, makes it feasible to to construct and maintain large programs. Imagine maintaining a one million line program without separate compilation. Any change to the source code would require a complete recompilation; the programmer would need to wait while one million lines of code compiled before testing a single line change. To make matters worse, all million lines would need to be consistent; this would make it difficult for multiple programmers to work simultaneously on different parts of the code.

The procedure provides three critical abstractions that allow programmers to construct non-trivial programs.

Control abstraction A procedure provides the programmer with a simple control abstraction; a standard mechanism exists for invoking a procedure and mapping it arguments, or parameters, into the called procedure's name space. A standard return mechanism allows the procedure to return control to the procedure that invoked it, continuing the execution of this "calling" procedure from the point immediately after the call. This standardization lets the compiler perform separate compilation.

Digression: A word about time

This chapter deals with both compile-time and run-time mechanisms. The distinction between events that occur at compile time and those that occur at run time can be confusing. All run-time actions are scripted at compile time; the compiler must understand the sequence of actions that will happen at run time to generate the instructions that cause the actions to occur. To gain that understanding, the compiler performs analysis at compile time and builds moderately complex compile-time structures that model the run-time behavior of the program. (See, for example, the discussion of lexically-scoped symbol tables in Section 6.7.3.) The compiler determines, at compile time, much of the storage layout that the program will use at run time; it then generates the code necessary to create that layout, to maintain it during execution, and to access variables and procedures in memory.

Name space Each procedure creates a new, protected name space; the programmer can declare new variables (and labels) without concern for conflicting declarations in other procedures. Inside the procedure, parameters can be referenced by their local names, rather than their external names. This lets the programmer write code that can be invoked in many different contexts.

External interface Procedures define the critical interfaces between the different parts of large software systems. The rules on name scoping, addressability, and orderly preservation of the run-time environment create a context in which the programmer can safely invoke code written by other individuals. This allows the use of libraries for graphical user interfaces, for scientific computation, and for access to system services.[1] In fact, the operating system uses the same interface to invoke an application program; it simply generates a call to some designated entry point, like `main`.

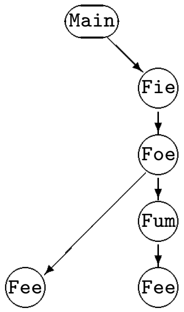
The procedure is, in many ways, the fundamental programming abstraction that underlies Algol-like languages. It is an elaborate facade created collaboratively by the compiler, the operating system software, and the underlying hardware. Procedures create named variables; the hardware understands a linear array of memory locations named with integer addresses. Procedures establish rules for visibility of names and addressability; the hardware typically provides several variants of a `load` and a `store` operation. Procedures let us decompose large software systems into components; these must be knit together into a complete program before the hardware can execute it, since the hardware simply advances its program counter through some sequence of individual instructions.

A large part of the compiler's task is putting in place the various pieces of the procedure abstraction. The compiler must dictate the layout of memory

[1] One of the original motivations for procedures was debugging. The user needed a known, correct mechanism to dump the contents of registers and memory after a program terminated abnormally. Keeping a `dump` routine in memory avoided the need to enter it through the console when it was needed.

```
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.
```

Call Tree



Execution History

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`

Figure 7.1: Non-recursive Pascal program

and encode that layout into the generated program. Since it may compile the different components of the program at different times, without knowing of their relationship to one another, this memory layout and all the conventions that it induces must be standardized and uniformly applied. The compiler must also understand the various interfaces provided by the operating system, to handle input and output, to manage memory, and to communicate with other processes.

This chapter focuses on the procedure as an abstraction and the mechanisms that the compiler uses to establish its control abstraction, its name space, and its interface to the outside world.

7.2 Control Abstraction

The procedure is, fundamentally, an abstraction that governs the transfer of control and the naming of data. This section explores the control aspects of procedure's behavior. The next section ties this behavior into the naming disciplines imposed in procedural languages.

In Algol-like languages, procedures have a simple and clear call/return discipline. On exit from a procedure, control returns to the point in the calling procedure that follows its invocation. If a procedure invokes other procedures, they return control in the same way. Figure 7.1 shows a Pascal program with several nested procedures. The *call tree* and *execution history* to its right summarize what happens when it executes. `Fee` is called twice: the first time from `Foe` and the second time from `Fum`. Each of these calls creates an instance, or an *invocation*, of `Fee`. By the time that `Fum` is called, the first instance of `Fee` is no longer active. It has returned control to `Foe`. Control cannot return to that instance of `Fee`; when `Fum` calls `Fee`, it creates a new instance of `Fee`.

The call tree makes these relationships explicit. It includes a distinct node for each invocation of a procedure. As the execution history shows, the only procedure invoked multiple times in the example is `Fee`. Accordingly, `Fee` has two distinct nodes in the call tree.

When the program executes the assignment `x := 1;` in the first invocation of `Fee`, the active procedures are `Fee`, `Foe`, `Fie`, and `Main`. These all lie on the path from the first instance of `Fee` to the program's entry in `Main`. Similarly, when it executes the second invocation of `Fee`, the active procedures are `Fee`, `Fum`, `Foe`, `Fie`, and `Main`. Again, they all lie on the path from the current procedure to `Main`.

The call and return mechanism used in Pascal ensures that all the currently active procedures lie along a single path through the call graph. Any procedure not on that path is uninteresting, in the sense that control cannot return to it. When it implements the call and return mechanism, the compiler must arrange to preserve enough information to allow the calls and returns to operate correctly. Thus, when `Foe` calls `Fum`, the calling mechanism must preserve the information needed to allow the return of control to `Foe`. (`Foe` may diverge, or not return, due to a run-time error, an infinite loop, or a call to another procedure that does not return.)

This simple call and return behavior can be modelled with a stack. As α calls β, it pushes the address for a return onto the stack. When β wants to return, it pops the address off the stack and branches to that address. If all procedures have followed the discipline, popping a return address off the stack exposes the next appropriate return address.

This mechanism is sufficient for our example, which lacks recursion. It works equally well for recursion. In a recursive program, the implementation must preserve a cyclic path through the call graph. The path must, however, have finite length—otherwise, the recursion never terminates. Stacking the return addresses has the effect of unrolling the path. A second call to procedure `Fum` would store a second return address in the location at the top of the stack—in

```
main() {
  printf("Fib(5) is %d.",
    fibonacci(5));
}

int fibonacci( ord )
  int ord;
{
  int one, two;
  if (ord < 1)
  {
    puts("Invalid input.");
    return ERROR_VALUE;
  }
  else if (ord == 1)
    return 0;
  else
    return fib(ord,&one,&two);
}
```

```
int fib(ord, f0, f1)
  int ord, *f0, *f1;
{
  int result, a, b;
  if (ord == 2)
  { /* base case */
    *f0 = 0;
    *f1 = 1;
    result = 1;
  }
  else
  { /* recurse */
    (void) fib(ord-1,&a,&b);
    result = a + b;
    *f0 = b;
    *f1 = result;
  }
  return result;
}
```

Figure 7.2: Recursion Example

effect, creating a distinct space to represent the second invocation of `Fum`. The same constraint applies to recursive and non-recursive calls: the stack needs enough space to represent the execution path.

To see this more clearly, consider the C program shown in Figure 7.2. It computes the fifth Fibonacci number using the classic recursive algorithm. When it executes, the routine `fibonacci` invokes `fib`, and `fib` invokes itself, recursively. This creates a series of calls:

Procedure	*Calls*
`main`	`fibonacci(5)`
`fibonacci`	`fib(5,*,*)`
`fib`	`fib(4,*,*)`
`fib`	`fib(3,*,*)`
`fib`	`fib(2,*,*)`

Here, the asterisk (`*`) indicates an uninitialized return parameter.

This series of calls has pushed five entries onto the control stack. The top three entries contain the address immediately after the call in `fib`. The next entry contains the address immediately after the call in `fibonacci`. The fourth entry contains the address immediately after the call to `fibonacci` in `main`.

After the final recursive call, denoted `fib(2,*,*)` above, `fib` executes the base case and the recursion unwinds. This produces a series of return actions:

Call	*Returns to*	*The result(s)*
fib(2,*,*)	fib(3,*,*)	1 (*one = 0; *two = 1;)
fib(3,*,*)	fib(4,*,*)	1 (*one = 1; *two = 1;)
fib(4,*,*)	fib(5,*,*)	2 (*one = 1; *two = 2;)
fib(5,*,*)	fibonacci(5)	3 (*one = 2; *two = 3;)
fibonacci(5)	main	3

The control stack correctly tracks these return addresses. This mechanism is sufficient for Pascal-style call and return. In fact, some computers have hard-wired this stack discipline into their call and return instructions.

More complex control flow Some programming languages allow a procedure to return a procedure and its run-time context. When the returned object is invoked, the procedure executes in the run-time context from which it was returned. A simple stack is inadequate to implement this control abstraction. Instead, the control information must be saved in some more general structure, such as a linked list, where traversing the structure does not imply deallocation. (See the discussion of heap allocation for activation records in the next section.)

7.3 Name Spaces

Most procedural languages provide the programmer with control over which procedures can read and write individual variables. A program will contain multiple name spaces; the rules that determine which statements can legally access each name space are called *scoping rules.*

7.3.1 Scoping Rules

Specific programming languages differ in the set of name spaces that they allow the programmer to create. Figure 7.3 summarizes the name scoping rules of several languages. Fortran, the oldest of these languages, creates two name spaces: a global space that contains the names of procedures and common blocks, and a separate name space inside each procedure. Names declared inside a procedure's local name space supersede global names for references within the procedure. Within a name space, different attributes can apply. For example, a local variable can be mentioned in a save statement. This has the effect of making the local variable a *static* variable—its value is preserved across calls to the procedure.

The programming language C has more complex scoping rules. It creates a global name space that holds all procedure names, as well as the names of global variables. It introduces a separate name space for all of the procedures in a single file (or compilation unit). Names in the file-level scope are declared with the attribute static; they are visible to any procedure in the file. The file-level scope holds both procedures and variables. Each procedure creates its own name space for variables and parameters. Inside a procedure, the programmer can create additional name spaces by opening a block (with { and }). A block can declare its own local names; it can also contain other blocks.

Fortran 77

Global Name Space
- Procedure Names
- Common Blocks
  - qualified variable names

Procedure Name Space
- Variables & Parameters

C

Global Name Space
- Procedures
- Global Variables

File Static Storage
- Procedures
- Variables

Procedure Name Space
- Variables & Parameters

Block Name Space
- Variables

PL/I

Global Name Space
- Variables
- Procedures

Procedure Name Space
- Variables & Parameters
- Procedures

Scheme

Global Name Space
- Objects
- Built-in objects

Local Name Space
- Objects

Figure 7.3: Name Space Structure of Different Languages

Pascal and PL/I have simple Algol-like scoping rules. As with Fortran, they have a global name space that holds procedures and variables. As with Fortran, each procedure creates its own local name space. Unlike Fortran, a procedure can contain the declarations of other procedures, with their own name spaces. This creates a set of nested lexical scopes. The example in Figure 7.1 does this, nesting `Fee` and `Fie` in `Main`, and `Fum` inside `Foe` inside `Fie`. Section 7.3.5 examines nested scopes in more detail.

7.3.2 Activation Records

The creation of a new, separate, name space is a critical part of the procedure abstraction. Inside a procedure, the programmer can declare named variables that are not accessible outside the procedure. These named variables may be initialized to known values. In Algol-like languages, local variables have lifetimes that match the procedures that declare them. Thus, they require storage during the lifetime of the invocation and their values are of interest only while the invocation that created them is active.

To accommodate this behavior, the compiler arranges to set aside a region of memory, called an *activation record* (AR), for each individual call to a procedure. The AR is allocated when the procedure is invoked; under most circumstances, it is freed when control returns from the procedure back to the point in the

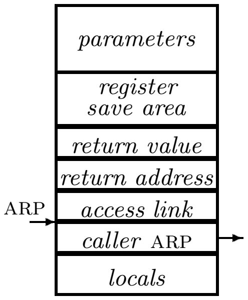


Figure 7.4: A typical activation record

program that called it. The AR includes all the storage required for the procedure's local variables and any other data needed to maintain the illusion of the procedure. Unless a separate hardware mechanism supports call and return, this state information includes the return address for this invocation of the procedure. Conveniently, this state information has the same lifetime as the local variables.

When *p* calls *q*, the code sequence that implements the call must both preserve *p*'s environment and create a new environment for *q*. All of the information required to accomplish this is stored in their respective ARs. Figure 7.4 shows how the contents of an AR might be laid out. It contains storage space for local variables, the AR pointer of the calling procedure, a pointer to provide access to variables in surrounding lexical scopes, a slot for the return address in the calling procedure, a slot for storing the returned value (or a pointer to it), an area for preserving register values on a call, and an area to hold parameter values. The entire record is addressed through an *activation record pointer*, denoted ARP. Taken together, the return address and the caller's ARP form a *control link* that allows the code to return control to the appropriate point in the calling procedure.

7.3.3 Local Storage

The activation record must hold all of the data and state information required for each invocation of a procedure *p*. A typical AR might be laid out as shown in Figure 7.4. The procedure accesses it AR through the ARP. (By convention, the ARP usually resides in a fixed register. In the ILOC examples, R_0 holds the ARP.) The ARP points to a designated location in the AR so that all accesses to the AR can be made relative to the ARP.

Items that need space in the AR include the ARP of the calling procedure, a return address, any registers saved in calls that the current procedure will make, and any parameters that will be passed to those procedures. The AR may also include space for a return value, if the procedure is a function, and an *access link* that can be used to find local variables of other active procedures.

Space for Local Data Each local data item may need space in the AR. The compiler should assign each location an appropriately-sized area, and record in the symbol table its offset from the ARP. Local variables can then be accessed as offsets from the ARP. In ILOC, this is accomplished with a `loadAO` instruction. The compiler may need to leave space among the local variables for pointers to variable-sized data, such as an array whose size is not known at compile time.

Space for Variable-length Data Sometimes, the compiler may not know the size of a data item at compile time. Perhaps its size is read from external media, or it must grow in response to the amount of data presented by other procedures. To accommodate such variable-sized data, the compiler leaves space for a pointer in the AR, and then allocates space elsewhere (either in the heap or on the end of the current AR) for the data once its size is known. If the register allocator is unable to keep the pointer in a register, this introduces an extra level of indirection into any memory reference for the variable-length data item. If the compiler allocates space for variable-length data in the AR, it may need to reserve an extra slot in the AR to hold a pointer to the end of the AR.

Initializing Variables A final part of AR maintenance involves initializing data. Many languages allow the programmer to specify a first, or initial, value for a variable. If the variable is allocated statically—that is, it has a lifetime that is independent of any procedure—the data can be inserted directly into the appropriate locations by the linker/loader. On the other hand, local variables must be initialized by executing instructions at run-time. Since a procedure may be invoked multiple times, and its ARs may be placed at different addresses, the only feasible way to set initial values is to generate instructions that store the necessary values to the appropriate locations. This code must run before the procedure's first executable statement.

Space for Saved Register Values When p calls q, either p or q must save the register values that p needs. This may include all the register values; it may be a subset of them. Whichever procedure saves these values must restore them after q's body finishes executing. This requires space, in the AR, for the saved registers. If the caller saves its own registers, then p's register save area will contain values related to its own execution. If the callee saves the caller's registers, then the value of p's registers will be preserved in q's register save area.

7.3.4 Allocating Activation Records

The compiler must arrange for an appropriately sized activation record for each invocation of a procedure. The activation record must be allocated space in memory. The space must be available when the calling procedure begins to execute the procedure call, so that it can fill in the various slots in the AR that contain information not known at compile-time, establishing both the control link and the access link. (Since a procedure can be called from many call sites, this information cannot, in general, be known before the invocation.) In general,

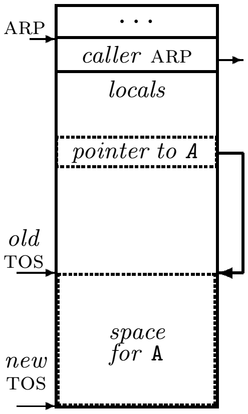


Figure 7.5: Allocating a dynamically sized array

the compiler has several choices for how to allocate activation records.

Stack Allocation of Activation Records When the contents of an AR are limited to the lifetime of the procedure invocation that caused its creation, and a called procedure cannot outlive its caller, the compiler can allocate ARs using a stack discipline. With these restrictions, calls and returns are balanced; each called procedure eventually returns, and any returns occurring between a procedure p's invocation and p's eventual return are the result (either directly or indirectly) of some procedure call made inside p. Stack allocation is attractive because the cost of both allocation and deallocation are small—a single arithmetic operation. (In contrast, general heap management schemes require much more work. See Section 7.7.)

Local variables whose sizes are not known at compile time can be handled within a stack allocation scheme. The compiler must arrange to allocate them at the end of the AR that matches the end of the stack. It must also keep an explicit pointer to the end of the stack. With these provisions, the running code can extend the AR and the stack to create space when the variable's size becomes known. To simplify addressing, the compiler may need to set aside a slot in the local variable area that holds a pointer to the actual data. Figure 7.5 shows the lower portion of an activation record where a dynamically-sized array, A, has been allocated at the end of the AR. The top of stack position (TOS) is shown both before and after the allocation of A.

With stack-allocated ARs, the AR for the currently executing procedure always resides at the top of the stack. Unless the size of A exceeds the available stack space, it can be added to the current AR. This allocation is inexpensive; the code can simply store TOS in the slot reserved for a pointer to A and increment TOS by the size of A. (Compare this to the cost of heap allocation and deallocation in Section 7.7.) Of course, when it generates code for this allocation, the compiler can insert a test that checks the size of A against available space. If insufficient stack space is available, it can either report an error or try another allocation mechanism. To allocate `A` on the heap would require the same pointer field for `A` in the AR. The code would simply use the standard heap management routines to allocate and free the space.

Heap Allocation of Activation Records If a procedure can outlive its caller, as in continuation-passing style, stack allocation is inappropriate because the caller's activation record cannot be freed. If a procedure can return an object that includes, explicitly or implicitly, references to its local variables, stack allocation is inappropriate because it will leave behind dangling pointers. In these situations, ARs can be kept in heap storage. This lets the code dismantle and free an AR when all the pointers to it have been discarded. Garbage collection is the natural technology for reclaiming ARs in such an environment, since the collector will track down all the pointers and ensure safety. (See Section 7.7.)

Static Allocation of Activation Records If a procedure q calls no other procedures, then q can never have more than a single active invocation. We call q a *leaf* procedure since it terminates a path through a graph of the possible procedure calls. The compiler can statically allocate activation records for leaf procedures. If the convention requires a caller to save its own registers, q will not need the corresponding space in its AR. This saves the run-time cost of AR allocation.

At any point in the execution of the compiled code, 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 leaf procedure. The static variables declared in any of the leaf procedures can be laid out together in that single AR. Using a single static AR for leaf procedures reduces the run-time space overhead of static AR allocation.

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 activation records. For example, if a call from `fee` to `fie` always results in calls to `foe` and `fum`, the compiler may find it profitable to allocate the ARs for `fie`, `foe`, and `fum` at the same time. If ARs are allocated on the heap, the savings are obvious. For stack-allocated ARs, the savings are minor.

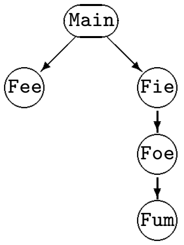
If all the calls to `fie` cannot be changed to allocate the coalesced AR, then the calls to `foe` and `fum` become more complex. The calling sequence generated must recognize when to allocate a new AR and when one already exists. The compiler must either insert conditional logic in the calling sequence to handle this, or it can generate two copies of the code for the affected procedures and call the appropriate routines. (The latter is a simple case of a transformation known as *procedure cloning*. See Chapter 14.)

```
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.
```

Name Resolution

Scope	x	y	z
Main	Main.x	Main.y	Main.z
Fee	Fee.x	Main.y	Main.z
Fie	Main.x	Fie.y	Main.z
Foe	Main.x	Fie.y	Foe.z
Fum	Main.x	Fum.y	Foe.z

Nesting Relationships



Calling Relationships

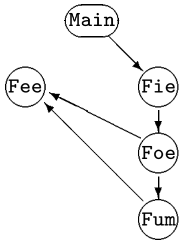


Figure 7.6: Nested lexical scopes in Pascal

7.3.5 Nested lexical scopes

Many Algol-like languages allow the programmer to define new name spaces, or lexical scopes, inside other scopes. The most common case is nesting procedure definitions inside one another; this approach is exemplified by Pascal. Any Pascal program that uses more than one procedure has nested scopes. C treats each new block, denoted by brackets { and }, as a distinct scope. Thus, the programmer can declare new variables inside each block.

Lexical scoping rules follow one general principle. Inside a given scope, names are bound to the lexically closest declaration for that name. Consider, again, our example Pascal program. It is shown, along with information about the nesting of lexical scopes, in Figure 7.6. It contains five distinct scopes, one corresponding to the program `Main` and one for each of the procedures `Fee`, `Fie`, Foe, and Fum. Each procedure declares some set of variables drawn from the set of names x, y, and z. Pascal's name scoping rules dictate that a reference to a variable is bound to the nearest declaration of that name, in lexical order. Thus, the statement `x := 1;` inside Fee refers to the integer variable x declared in Fee. Since Fee does not declare y, the next statement refers to the integer variable y declared in Main. In Pascal, a statement can only "see" variables declared in its procedure, or in some procedure that contains its procedure. Thus, the assignment to x in Fum refers to the integer variable declared in Main, which contains Fum. Similarly, the reference to z in the same statement refers to the variable declared in Foe. When Fum calls Fee, Fum does not have access to the value of x computed by Fee. These scoping relationships are detailed in the table on the right-hand side of Figure 7.6. The nesting relationships are depicted graphically immediately below the table. The bottom graph shows the sequence of procedure calls that occur as the program executes.

Most Algol-like languages implement multiple scopes. Some, like Pascal, PL/I, and Scheme, allow arbitrary nesting of lexical scopes. Others have a small, constant set of scopes. Fortran, for example, implements a global name space and a separate name space for each subroutine or function. (Features such as block data, statement functions, and multiple-entry procedures slightly complicate this picture.)

C implements nested scopes for blocks, but not for procedures. It uses the static scope, which occurs at the file-level, to create a modicum of the modularity and information hiding capabilities allowed by Pascal-style nested procedures. Since blocks lack both parameter binding and the control abstraction of procedures, the actual use of nested scopes is far more limited in C. One common use for the block-level scope is to create local temporary storage for code generated by expanding a pre-processor macro.

In general, each scope corresponds to a different region in memory, sometimes called a *data area*. Thus, the data area in a procedure's activation record holds its locally-declared variables. Global variables are stored in a data-area that can be addressed by all procedures.

To handle references in languages that use nested lexical scopes, the compiler translates each name that refers to a local variable of some procedure into a pair ⟨*level,offset*⟩, where *level* is the lexical nesting level of the scope that declared the variable and *offset* is its memory location relative to ARP. This pair is the variable's *static distance coordinate*. The translation is typically done during parsing, using a lexically-scoped symbol table, described in Section 6.7.3. The static distance coordinate encodes all the information needed by the code generator to emit code that locates the value at run time. The code generator must establish a mechanism for finding the appropriate AR given *level*. It then emits code to load the value found at *offset* inside that AR. Section 7.5.2 describes two different run-time data structures that can accomplish this task.

7.4 Communicating 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 parameters. To use the operation, the programmer invokes the procedure with an appropriate *binding* of values to those parameters. The called procedure also needs a mechanism to return its result. If it produces a result suitable for assignment, we say that it *returns* a value and we call the procedure a *function* rather than a plain procedure.

7.4.1 Passing Parameters

Parameter naming lets the programmer write the procedure in terms of its local name space, and then invoke it in many different contexts to operate on many different arguments. This notion of mapping arguments from the call into the procedure's name space is critical to our ability to write abstract, modular codes.

Most modern programming languages use one of two rules for mapping actual parameters at a call site into the formal parameters declared inside the called procedure, *call-by-value* binding and *call-by-reference* binding. While these techniques differ in their behavior, the distinction between them is best explained by understanding their implementation. Consider the following procedure, written in C, and several call sites that invoke it.

```
int fee(x,y)          c = fee(2,3);
  int x,y;            a = 2;
{                     b = 3;
  x = 2 * x;          c = fee(a,b);
  y = x + y;          a = 2;
  return y;           b = 3;
}                     c = fee(a,a);
```

With call-by-value, as in C, the calling procedure creates a location for the formal parameter in the called procedure's AR and copies the value of the actual parameter into that location. The formal parameter has its own storage and its own value. The only way to modify its value is by referring directly to its name. The sole constraint placed on it is its initial value, which is determined by evaluating the actual parameter at the time of the call.

The three different invocations produce the same results under call-by-value.

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

Because the actual parameters and formal parameters have the same value, rather than the same address, the behavior is both intuitive and consistent. None of the calls changes the value of either a or b.

Digression: Call-by-Name Parameter Binding

Algol introduced the notion of *call-by-name* parameter binding. Call-by-name has a simple meaning. A reference to the formal parameter behaves exactly as if the actual parameter had been textually substituted in place of the formal. This can lead to some complex and interesting behavior. Consider the following short example in Algol-60:

```
begin comment Call-by-Name example;

  procedure zero(Arr,i,j,u1,u2);
    integer Arr;
    integer 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, z;

  x := 100;
  y := 200;

  zero(Work[p,q],p,q,x,y);
end
```

The procedure `zero` assigns the value 0 to every element of the array `Work`. To see this, rewrite `zero` with the text of the actual parameters.

This elegant idea fell from use because it was difficult to implement. In general, each parameter must be compiled into a small function of the formal parameters that returns a pointer. These functions are called *thunks*. Generating competent thunks was complex; evaluating thunks for each access to a parameter raised the cost of parameter access. In the end, these disadvantages overcame any extra generality or transparency that the simple rewriting semantics offered.

In a language that uses call-by-reference parameter passing, the calling procedure creates a location for a pointer to the formal parameter and fills that location with a pointer to the result of evaluating the expression. Inside the called procedure, references to the formal parameter involve an extra level of indirection to reach the value. This has two critical consequences. First, any redefinition of the call-by-reference formal parameter is reflected in the actual parameter, unless the actual parameter is an expression rather than a variable reference. Second, any call-by-reference formal parameter might be bound to a variable that is accessible by another name inside the called procedure.

The same example, rewritten in PL/I, produces different results because of the call-by-reference parameter binding.

```
procedure fee(x,y)
  returns fixed binary;
  declare x,y fixed binary;
  begin;
    x = 2 * x;
    y = x + y;
    return y;
    end;
```

```
c = fee(2,3);
a = 2;
b = 3;
c = fee(a,b);
a = 2;
b = 3;
c = fee(a,a);
```

PL/I's call-by-reference parameter passing produces different results than the C code did.

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

Notice that the second call redefines both a and b; the behavior of call-by-reference is intended to communicate changes made in the called procedure back to the calling environment. The third call creates an alias between x and y in fee. The first statement redefines a to have the value 4. The next statement references the value of a twice, and adds the value of a to itself. This causes fee to return the value 8, rather than 6.

In both call-by-value and call-by-reference, the space requirements for representing parameters are small. Since the representation of each parameter must be copied into the AR of the called procedure on each call, this has an impact on the cost of the call. To pass a large object, most languages use call-by-reference for efficiency. For example, in C, the string representation passes a fixed-size pointer rather than the text. C programmers typically pass a pointer to the array rather than copying each element's value on each call.

Some Fortran compilers have used an alternative binding mechanism known as *call-by-value/result.* The call operates as in call-by-value binding; on exit, values from the formal parameters are copied back to the actual parameters—except when the actual is an expression.

Call-by-value/result produces the same results as call-by-reference, unless the call site introduces an alias between two or more formal parameters. (See Chapter 13 for a deeper discussion of parameter-based aliasing.) The call-by-value/result binding scheme makes each reference in the called procedure cheaper, since it avoids the extra indirection. It adds the cost of copying values in and out of the called procedure on each call.

Our example, recoded in Fortran, looks like:

```
integer function fee(x,y)
  integer x, y
  x = 2 * x
  y = x + y
  return y
  end
```

```
c = fee(2,3)
a = 2
b = 3
c = fee(a,b)
a = 2
b = 3
c = fee(a,a)
```

Since the Fortran standard forbids aliasing that involves formal parameters, this program is not a legal Fortran program. The standard allows the compiler to interpret a non-standard conforming program in any convenient way. However, most Fortran compilers will neither detect this particular problem, nor implement it incorrectly.

A Fortran version of our program, implemented with value/result, would produce the following behavior.

Call by	a		b		*Return*
Value/Result	*in*	*out*	*in*	*out*	*Value*
fee(2,3)	–	–	–	–	7
fee(a,b)	2	4	3	7	7
fee(a,a)	2	*	3	3	6

Note that, for the third call site, the value for a after the call is dependent on the call-by-value/result implementation. Depending on the order of assignment to a, a could have the value 6 or 4.

7.4.2 Returning Values

To return a value for the function, as opposed to changing the value of one of its actual parameters, the compiler must set aside space for the returned value. Because the return value, by definition, is used after the called procedure terminates, it needs storage outside the called procedure's AR. If the compiler-writer can ensure that the return value is of small, fixed size, then it can store the value in either the AR of the calling procedure (at a fixed offset) or in a register.

To achieve this goal, the compiler can allocate a fixed slot in the AR of the calling procedure or a designated hardware register, and use that slot to hold a pointer to the actual value. This allows the called routine to return an arbitrarily-sized value to the caller; space for the return value is allocated in the caller's AR prior to the call and the appropriate pointer is stored in the return value slot of the caller's AR. To return the value, the called procedure loads the pointer from ARP + *offset*(*return value*), and uses it to store the return value.

This scenario can be improved. If the return value is a small, simple value such as a simple integer or a floating-point number, it can be returned in the slot allocated for the pointer. As long as both the caller and the callee know the type of the returned value, the compiler can safely and correctly eliminate the indirection.

7.5 Establishing Addressability

As part of the linkage convention, the compiler must ensure that each procedure can generate an address for each variable that it references. In an Algol-like language, this usually includes named global variables, some form of static storage, the procedure's own local variables, and some of the local variables of its lexical ancestors. In general two cases arise; they differ in the amount of calculation required to find the starting address, or *base address*, of the data area.

7.5.1 Trivial Base Addresses

For most variables, the compiler can emit code that generates the base address in one or two instructions. The easiest case is a local variable of the current procedure. If the variable is stored in the procedure's AR, the compiler can use the ARP as its base address. The compiler can load the variable's value with a single loadAI instruction or a loadI followed by a loadAO. Thus, access to local variables is fast.

(Sometimes, a local variable is not stored in the procedure's AR. The value might reside in a register, in which case loads and stores are not needed. The variable might have an unpredictable or changing size, in which case the compiler might need to allocate space for it in the run-time heap. In this case, the compiler would likely reserve space in the AR for a pointer to the heap location. This adds one extra level of indirection to any access for that variable, but defers the need for its size until run-time.)

Access to global and static variables is handled similarly, except that the base address may not be in a register at the point where the access occurs. Thus, the compiler may need to emit code that determines the base address at run-time. While that sounds complex, it is exactly the task that symbolic assemblers were designed to accomplish. The compiler generates base addresses for global and static data areas by using the name of the data area as part of an assembly language label. To avoid conflicts with other labels, the compiler "mangles" the name by adding a prefix, a suffix, or both, to the name. The compiler deliberately adds characters that cannot appear in source-language names.

For example, given a global variable fee, a C compiler might construct the label &fee., counting on the fact that ampersand (&) cannot be used in a source language name and that no legal C name can end with a period. It would emit the appropriate assembly language pseudo-operation to reserve space for fee or to initialize fee, attaching the label to the pseudo-operation. To obtain a run-time address for fee, the compiler would emit the instruction

$$\texttt{loadI}\quad \texttt{\&fee.}\quad \Rightarrow \texttt{r}_1.$$

The next instruction would use $\texttt{r}_1$ to access the memory location for fee.

Notice that the compiler has no actual knowledge of where fee is stored. It uses a relocatable label to ensure that the appropriate run-time address is written into the instruction stream. At compile-time, it makes the link between the contents of $\texttt{r}_1$ and the location of `fee` by creating an assembly-level label. That link is resolved by the operating system's loader when the program is loaded and launched.

Global variables may be labelled 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` instructions relative to the common block's label.

Similarly, the compiler may create a single static data area for all of the static variables within a single static scope. This serves two purposes. First, it keeps the set of labels smaller, decreasing the likelihood of an unexpected conflict. If a name conflict occurs, it will be discovered during linking or loading. When this occurs, it can be quite confusing to the programmer. To further decrease the likelihood of this problem, the compiler can prepend part of the file name or the procedure name to the variable's name. Second, it decreases the number of base addresses that might be required in a single procedure. This reduces the number of registers tied up to hold base addresses. Using too many registers for addressing may adversely affect overall run-time performance of the compiled code.

7.5.2 Local Variables of Other Procedures

In a language that supports nested lexical scopes, the compiler must provide a mechanism to map static distance coordinates into hardware addresses for the corresponding variables. To accomplish this, the compiler must put in place data structures that let it to compute the addresses of ARs of each lexical ancestors of the current procedure.

For example, assume that `fee`, at level x, references variable `a` declared in `fee`'s level y ancestor `fie`. The parser converts this reference into a static distance coordinate $\langle (x-y), \mathit{offset}\rangle$. Here, $x-y$ specifies how many lexical levels lie between `fee` and `fie`, and *offset* is the distance from the ARP for an instance of `fie` and the storage reserved for `a` in `fie`'s AR.

To convert $\langle (x-y), \mathit{offset}\rangle$ into a run-time address, the compiler must emit two different kinds of code. First, the compiler writer must select a mechanism for tracking lexical ancestry among activation records. The compiler must emit the code necessary to keep this information current at each procedure call. Second, the compiler must emit, at the point of reference, code that will interpret the run-time data structure and the expression $x-y$ to produce the address of the appropriate ARP and use that ARP and *offset* to address the variable. Since both $x-y$ and *offset* are known at compile time, most of the run-time overhead goes into traversing the data structure.

Several mechanisms have been used to solve this problem. We will examine two: access links and a global display.

Access Links In this scheme, the compiler ensures that each AR contains a pointer to the AR of its immediate lexical ancestor. We call this pointer an *access link*, since it is used to access non-local variables. Starting with the

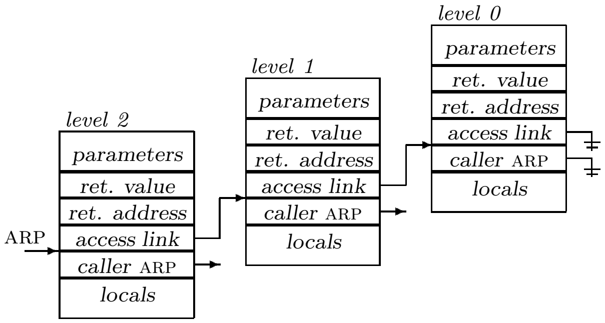


Figure 7.7: Using access links

current procedure, the access links form a chain of the ARs for all of its lexical ancestors. Any local variable of another procedure that can be accessed from the current procedure must be stored in an AR on the chain of access links. Figure 7.7 shows this situation.

To use access links, the compiler emits code that walks the chain of links until it finds the appropriate ARP. If the current procedure is at level x, and the reference is to offset o at level y, the compiler emits code to follow $x - y$ pointers in the chain of access links. This yields the appropriate ARP. Next, it emits code to add the offset o to this ARP, and to use the resulting address for the memory access. With this scheme, the cost of the address calculation is proportional to $x - y$. If programs exhibit shallow levels of lexical nesting, the difference in cost between accessing two variables at different levels will be fairly small. Of course, as memory latencies rise, the constant in this asymptotic equation gets larger.

To maintain access links, the compiler must add code to each procedure call to find the appropriate ARP and store it into the AR for the called procedure. Two cases arise. If the called procedure is nested inside the current procedure—that is, its lexical level is exactly one more than the level of the calling procedure—then the caller uses its own ARP as the access link of the called procedure. Otherwise, the lexical level must be less than or equal to the level of the calling procedure. To find the appropriate ARP, the compiler emits code to find the ARP one level above the called procedure's level. It uses the same mechanism used in accessing a variable at that level; it walks the chain of access links. It stores this ARP as the called procedure's access link.

Global Display In this scheme, the compiler allocates a globally accessible array to hold the ARPs of the most recent instance of a procedure called at each level. Any reference to a variable that resides in some lexical ancestor becomes an indirect reference through this global table of ARPs. To convert $\langle (x - y), \mathit{offset} \rangle$ into an address, the compiler takes the ARP stored in element y of the global display, adds *offset* to it, and uses that as the address for the memory reference.

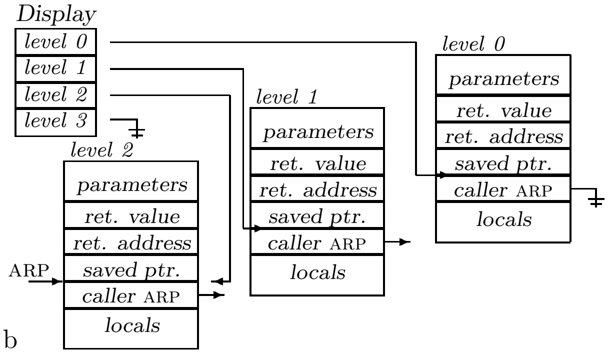


Figure 7.8: Using a display

Figure 7.8 shows this situation.

Using a global display, the cost of non-local access is independent of $x - y$. The compiler knows y, so the overhead consists of a minor address calculation (see Section 8.5) and an extra load operation. This still leaves an inequity between the cost of local access and the cost of non-local access, but the difference is smaller than with access links and it is entirely predictable.

To maintain a global display, the compiler must add code to both the procedure call and its corresponding return. On call, the procedure should store the display entry for its lexical level in its AR, and replace that entry in the global display with its own ARP. On return, it should restore the value from its AR to the global display. This simple scheme takes one more slot in the global display than is strictly necessary, but this is a small price to pay for the simplicity of the update scheme.

As an improvement, the compiler can omit the code to update the display in any procedure that, itself, calls no other procedures. This eliminates some of the wasted display updates. It does not eliminate all of them. If the procedure never calls a procedure that is more deeply nested, then it can omit the display update, because execution can only reach a more deeply nested procedure by passing through some intervening procedure that calls down the lexical nesting tree and will, therefore, preserve the appropriate ARP.

7.6 Standardized Linkages

The procedure linkage is a social contract between the compiler, the operating system, and the target machine that clearly divides responsibility for naming, for allocation of resources, for addressability, and for protection. The procedure linkage ensures interoperability of procedures between the user's code, as translated by the compiler, and the operating system's routines. Typically, all of the compilers for a given combination of target machine and operating system use the same procedure linkage.

The linkage convention serves to isolate each procedure from the different

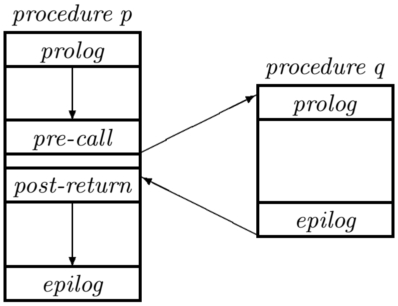


Figure 7.9: A standard procedure linkage

environments found at call sites that invoke it. Assume that procedure `p` has an integer parameter `x`. Different calls to `p` may bind `x` to a local variable stored in the calling procedure's stack frame, to a global variable, to an element of some static array, and to the result of evaluating an integer expression such as `y+2`. Because the procedure linkage specifies how to evaluate and store `x` in the calling procedure, and how to access `x` in the called procedure, the compiler can generate code for the body of the called procedure that ignores the differences between the run-time environments at the different calls to `p`. As long as all the procedures obey the linkage convention, the details will mesh together to create the seamless transfer of values promised by the source-language specification.

The linkage convention is, of necessity, machine dependent. For example, the linkage convention implicitly contains information such as the number of registers available on the target machine, and the mechanism for executing a call and a return.

Figures 7.9 shows how the pieces of a standard procedure linkage fit together. Each procedure has a *prolog sequence* and an *epilog sequence*. Each call site includes both a *pre-call sequence* and a *post-return sequence*.

pre-call The pre-call sequence is responsible for setting up the called procedure's AR. It allocates space for the basic AR and fills in each of the locations. This includes evaluating each actual parameter to a value or an address as appropriate, and storing the value in the parameter's designated slot in the AR. It also stores the return address, the calling procedure's ARP, and, if necessary, the address of the space reserved for a return value in their designated locations in the AR.

The caller cannot know how much space to reserve for the called procedure's local variables. Thus, it can only allocate space for the basic portion of the called procedure's AR. The remaining space must be allocated by the called procedure's prolog code.[2]

[2] The alternative is to arrange a link-time mechanism to place this information in a place where the caller can find it. If ARs are stack allocated, extending the AR in the called procedure

post-return The post-call sequence must undo the actions of the pre-call sequence. It completes the process by deallocating the called procedure's AR. If any call-by-reference parameters need to be returned to registers, the post-return sequence restores them.

prolog Each procedure's prolog performs the same basic set of actions. It completes the task of constructing the called procedure's run-time environment. The prolog extends the procedure's basic AR to create space for local variables, and initializes them as necessary. If the procedure contains references to a procedure-specific static data area, the prolog may need to establish addressability by loading the appropriate label into a register.

If the compiler is using a display to access local-variables of other procedures, the prolog updates the display entry for its lexical level. This involves saving the current entry for that level into the AR and storing the current ARP into the display.

epilog A procedure's epilog code undoes some of the actions of the prolog. It may need to deallocate the space used for local variables. It must restore the caller's ARP and jump to the return address. If the procedure returns a value, that value is actually stored by code generated for the return statement (whether the return is explicit or implicit).

This is, however, just a general framework for building the linkage convention.

Figure 7.10 shows one possible division of labor between the caller and the callee. It includes most of the details that the linkage convention must handle. It does not mention either a display or access links.

- To manage a display, the prolog sequence in the called procedure saves the current display record for its own level into its AR and stores its own ARP into that slot in the display. This establishes the display pointer for any call that it makes to a more deeply nested procedure. If the procedure makes no calls, or only calls less nested procedures, it can skip this step.
- To manage access links, the pre-call sequence in the calling procedure computes the appropriate first access link for the called procedure and saves it into the access link slot in the called procedure's AR. This can be the caller's own ARP (if the callee is nested inside the caller), the caller's access link (if the callee is at the same lexical level as the caller), or some link up the caller's access link chain (if the callee is declared at an outer lexical level).

As long as the called procedure is known at compile time, maintaining either a display or access links is reasonably efficient.

One critical issue in the procedure linkage is preserving values kept in registers. This can be done by saving them in the pre-call sequence and restoring

is easy. If ARs are allocated in the heap, the compiler writer may elect to use a separately allocated block of memory to hold the local variables.

	Caller	Callee
Call	*pre-call sequence*	*prolog sequence*
	allocate basic AR	preserve callee-save registers
	evaluate & store parameters	extend AR for local data
	store return address & ARP	find static data area
	save caller-save registers	initialize locals
	jump to callee	fall through to code
Return	*post-return sequence*	*epilog sequence*
	deallocate basic AR	restore callee-save registers
	restore caller-save registers	discard local data
	restore reference parameters	restore caller's ARP
		jump to return address

Figure 7.10: One possible division of responsibility in a linkage

them in the post-call sequence; this convention is called *callee-saves*. The alternative is to save them in the prolog sequence and restore them in the epilog sequence; this convention is called *caller-saves*. With callee saves, the enregistered values are stored in the calling procedure's AR. With caller saves, the enregistered values are stored in the called procedure's AR.

Each convention has arguments in its favor. The procedure that saves and restores registers needs only to preserve a subset of the enregistered values. In caller saves, the pre-call sequence only needs to save a value if it is used after the call. Similarly, in callee saves, the prolog only needs to save a value if the procedure actually uses the register that contains it. The linkage convention can specify that all registers are caller-saves, that all registers are callee-saves, or it can divide the register set into some caller-saves and some callee-saves registers. For any specific division of responsibility, we can construct programs where the division fits well and programs where it does not. Many modern systems divide the register set evenly between these two conventions.

7.7 Managing Memory

Another issue that the compiler writer must face in implementing procedures is memory management. In most modern systems, each program executes in its own logical address space. The layout, organization, and management of this address space requires cooperation between the compiler and the operating system to provide an efficient implementation that falls within rules and restrictions imposed by the source language and the target machine.

7.7.1 Memory Layout

The compiler, the operating system, and the target machine cooperate to ensure that multiple programs can execute safely on an interleaved (time-sliced) basis.

Digression: More about time

In a typical system, the linkage convention is negotiated between the compiler implementors and the operating system implementors at an early stage in 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 the procedure prolog and epilog sequences for each procedure, along with the pre-call and post-call sequences for each call site. This code executes at run time. Thus, 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, put in place a mechanism that will generate that address at either link-time (using a relocatable assembly-language label) or at run-time (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 single global display to provide addressability for local variables of other procedures, the compiler cannot know the run-time addresses of the appropriate ARs. 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 arranged by using a relocatable assembly-language label. Thus, the prolog code can simply store the current display entry for the procedure's level into its AR (using a `loadAO` from the display address) and store it into the AR (using a `storeAO` from the frame pointer).

Many of the decisions about how to layout, manipulate, and manage the program's address space lie outside the purview of the compiler writer. However, the decisions have a strong impact on the code that must be generated and the performance achieved by that code. Thus, the compiler writer must have a broad understanding of these issues.

Placing Run-time Data Structures At run-time, a compiled program consists of executable code and several distinct categories of data. The compiled code is, in general, fixed in size. Some of the data areas are also fixed in size; for example, the data areas for global and static variables in languages like Fortran and C neither grow nor shrink during execution. Other data areas have sizes that change throughout execution; for example, the area that holds ARs for active procedures will both expand and shrink as the program executes.

Figure 7.11 shows a typical layout for the address space used by a single compiled program. The executable code sits at the low end of the address space; the adjacent region, labelled *Static* in the diagram, holds both static and global data areas. The remainder of the address space is devoted to data areas that expand and contract; if the language allows stack allocation of ARs, the compiler needs to leave space for both the heap and the stack. To allow best utilization of the space, they should be placed at opposite ends of the open space and allowed to grow towards each other. The heap grows toward higher

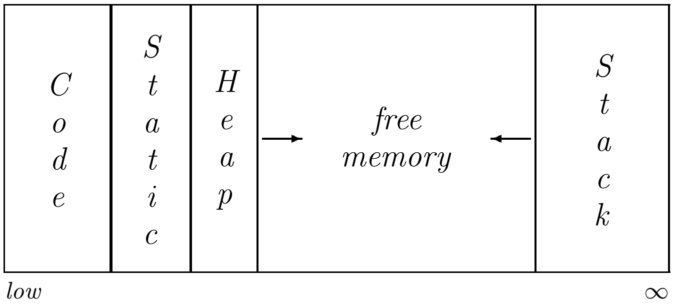


Figure 7.11: Logical address-space layout

addresses; the stack grows toward lower addresses. If activation records are kept on the heap, the run-time stack may be unneeded.

From the compiler's perspective, the logical address space is the whole picture. However, modern computer systems typically execute many programs in an interleaved fashion. The operating system maps several different logical address spaces into the single address space supported by the target machine. Figure 7.12 shows this larger picture. Each program is isolated in its own logical address space; each can behave as if it has its own machine.

A single logical address space can be spread across disjoint segments (or pages) of the physical address space; thus, the addresses 100,000 and 200,000 in the program's logical address space need not be 100,000 bytes apart in physical memory. In fact, the physical address associated with the logical address 100,000 may be larger than the physical address associated with the logical address 200,000. The mapping from logical addresses to physical addresses is maintained cooperatively by the hardware and the operating system. It is, in large part, beyond the compiler's purview.

Impact of Memory Model on Code Shape The compiler writer must decide whether to keep values aggressively in registers, or to keep them in memory. This decision has a major impact on the code that the compiler emits for individual statements.

With a memory-to-memory model, the compiler typically works within the limited set of registers on the target machine. The code that it emits uses real register names. The compiler ensures, on a statement-by-statement basis, that demand for registers does not exceed the set of registers available on the target machine. Under these circumstances, register allocation becomes an optimization that improves the code, rather than a transformation that is necessary for correctness.

With a register-to-register model, the compiler typically works with a set of virtual registers, rather than the real register set of the target machine. The virtual register set has unlimited size. The compiler associates a virtual register

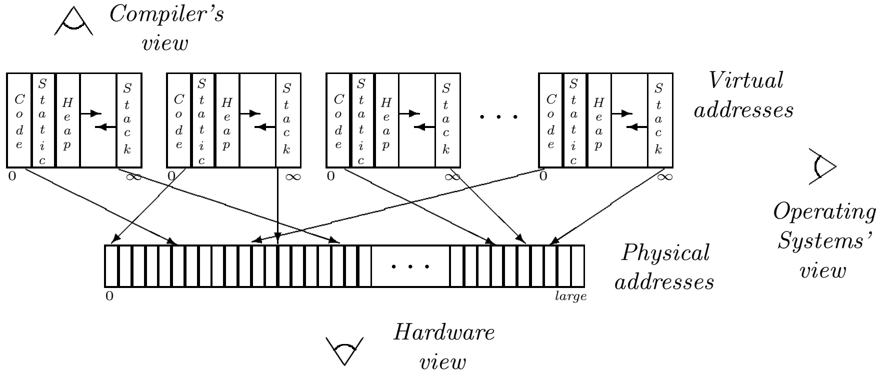


Figure 7.12: Different views of the address space

with each value that can legally reside in a register;[3] such a value is stored to memory only when it is passed as a call-by-reference parameter, when it is passed as a return value, or when the register allocator spills it (see Chapter 10. With a register-to-register memory model, the register allocator must be run to reduce demand for registers and to map the virtual register names into target machine register names.

Alignment and Padding Target machines have specific requirements on where data items can be stored. A typical set of restrictions might specify that 32-bit integers and 32-bit floating-point numbers begin on a full-word boundary (32-bit), that 64-bit floating-point data begin on a double-word (64-bit) boundary, and that string data begin on a half-word (16-bit) boundary. We call these rules *alignment rules.*

Some machines have a specific instruction to implement procedure calls; it might save registers or store the return address. Such support can add further restrictions; for example, the instruction might dictate some portions of the AR format and add an alignment rule for the start of each AR. The DEC VAX computers had a particularly elaborate call instruction; it took a 32-bit argument that specified which registers to save on call and restore on return.

To comply with the target machine's alignment rules, the compiler may need to waste some space. To assign locations in a data area, the compiler should order the variables into groups, from those with the most restrictive alignment rules to those with the least. (For example, double-word alignment is more restrictive than full-word alignment.) Assuming that the data area begins on a full-word boundary, it can place single-word variables at the start of the data area until it reaches an address that satisfies the most restrictive alignment rule. Then, it can place all the data in consecutive locations, without padding. It can

[3] In general, a value can be kept in a register if it can only be accessed using a single name. We call such a value an unambiguous value.

Digression: A Primer on Cache Memories

One way that architects try to bridge the gap between processor speed and memory speed is through 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 series of equal-sized *frames.* Each frame has an address field, called its *tag*, that holds a main memory address.

The hardware automatically maps memory locations into 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. The set of blocks that map to a given frame is called a *line.* 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 access to memory, the hardware checks to see if the requested block is already in its cache frame. If so, the requested bytes are returned to the processor. If not, the block currently in the frame is evicted and the requested block is brought into the cache.

Some caches use more complex mappings. A set-associative cache uses multiple frames for each cache line, typically two or four frames per line. A fully-associative cache can place any block in any frame. Both these use an associative search over the tags to determine if a block is in cache. Associative schemes a policy to determine which block to evict; common schemes are random replacement and least-recently used (LRU) 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 percentage of accesses that hit in the cache. From the compiler's perspective, the first three are fixed. Compiler-based efforts to improve memory performance focus on increasing the hit ratio and on ensuring that blocks are in the cache when needed.

assign all the variables in the most restricted category, followed by the next most restricted class, and so on, until all variables have offsets. Since alignment rules almost always specify a power of two, the end of one category will naturally fit the restriction for the next category. This scheme inserts padding if and only if the number of full-word variables available to it is less than the difference between the alignment of the word that begins the data area and the size of the most restricted group of variables.

Relative Offsets and Cache Performance The widespread use of cache memories in modern computer systems has subtle implications for the layout of variables in memory. If two values are used in near 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. This would guarantee that the values are fetched from RAM to cache together and that the impact of their presence in cache on other variables is minimized. If this cannot be arranged, the compiler would like to ensure that the two variables map into different cache lines—that is, the distance between their two addresses is not a multiple of the cache size divided by the cache line size.

If we consider just two variables, this issue seems quite manageable. When all active variables are considered, however, the problem can become complex. Most 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 non-interference 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 earlier, the mapping of the program's logical address space onto the hardware's physical address space need not preserve the distance between them. 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 physical cache may use either virtual addresses or physical addresses in its tag fields. A virtual-address cache preserves the spacing between values that the compiler creates; with such a cache, the compiler may be able to plan non-interference between large objects. With a physical-address cache, the distance between two locations in different pages is determined by the page mapping (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 getting objects that are referenced together into the same page.

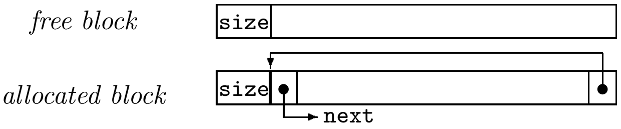
7.7.2 Algorithms for Managing the Heap

Many programming languages deal with objects that are dynamically created and destroyed. The compiler cannot determine the size or lifetime of these objects. To handle such objects, the compiler and the operating system create a pool of dynamically allocatable storage that is commonly called the *run-time heap*, or just the heap. Many issues arise in creating and managing the heap; some of these are exposed at the programming language level, while others are only visible to the authors of system software.

This section briefly explores the algorithms used to manage the heap, and some of the tradeoffs that can arise in implementing programming language interfaces to the heap management routines. We assume a simple interface to the heap: a routine `allocate(size)` and a routine `free(address)`. `Allocate` takes an integer argument `size` and returns the address of a block of space in the heap that contains at least `size` bytes. `Free` 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 heap management algorithms are (1) the speed of both `allocate` and `free`, (2) the extent to which the pool of free space becomes fragmented into small blocks, and (3) the necessity of using explicit calls to `free`. To introduce these issues, first consider a simple allocation model, called *first-fit allocation*.

First-fit Allocation The goal of a first-fit allocator is to create fast versions of `allocate` and `free`. As book-keeping overhead, every block in the heap has a hidden field that holds its size. In general, the size field is located in the word preceding the address returned by `allocate`. Blocks available for allocation reside on a list called the *free list*. In addition to the mandatory size field, each block on the free list has a pointer to the next block on the free list (or null), and a pointer to the block itself in the last word of the block.



The initial condition for the heap is a single large block placed on the algorithm's free list.

A call to `allocate(`k`)` causes the following sequence of events. `Allocate` 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 it finds an appropriate block, b_i. If b_i is larger than necessary, `allocate` creates a new free block from the excess space at the end of b$_i$ and places that block on the free list. `Allocate` returns a pointer to the second word of b_i.

If `allocate` fails to find a large enough block, it tries to extend the heap. If it succeeds in extending the heap, it returns a block of appropriate size from this newly allocated portion of the heap. If extending the heap fails, `allocate` reports the failure (typically by returning a null pointer).

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 produces a fast and simple `free` routine. 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 blocks that are free. `Free` can load the word preceding b_j's size field. If it is a valid pointer, and it points to a matching block header (one that points back to the start of b_j), then b_j can be added to the predecessor block by increasing its size field and storing the appropriate pointer in the last word of b_j. This action requires no update of the free list.

To combine b_j with its successor in memory, the `free` routine can use its size field to locate the next block. Using the successor's size field, it can determine if the end of that block points back to its header. If so, `free` can combine the two blocks, leaving b_j on the free list. Updating the free list is a little more complex. To make it efficient, the free list needs to be doubly linked. Of course, the pointers are stored in unallocated blocks, so the space overhead is irrelevant. Extra time required to update the doubly-linked free list is minimal.

Digression: Arena-based Allocation

Inside the compiler itself, the compiler writer may find it profitable to use a specialized allocator. Compilers have phase-oriented activity. This lends itself well to an arena-based allocation scheme [35].

With an arena-based allocator, the program creates an arena 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; instead, the entire arena is freed at once.

The arena-based allocator is a compromise between traditional allocators and garbage collecting allocators. With an arena-based allocator, the calls to `allocate` can be made lightweight (as in the modern allocator). No calls to free 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 this scheme have been tried. They tradeoff 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. Knuth has an excellent section describing allocation schemes similar to first fit [37, § 2.5].

Modern Allocators Modern allocators use a simple technique derived from first fit allocation, but simplified by a couple of 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.

Several modern allocators capitalize on these observations. They have separate memory pools for several common sizes. Typically, sizes are selected as powers of two, starting with a reasonably small size (such as sixteen bytes) and running up to the size of a virtual memory page (typically 2048 or 4096 bytes). Each pool has only one size 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.

These changes make both `allocate` and `free` quite fast. `Allocate` must check for an empty free list and increase the appropriate pool by a page if it is empty. `Free` simply inserts the 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 placing a size marker for all the blocks in the entire page in the first word on the page.

7.7.3 Implicit Deallocation

Many programming languages specify that the implementation will implicitly deallocate memory objects when they are no longer in use. This requires some care in the implementation of both the allocator and the compiled code. To perform implicit deallocation, the compiler and run-time system must include a mechanism for determining when an object is no longer of interest, or *dead*, and a mechanism for reclaiming and recycling that dead data.

The two classic techniques for implicit deallocation are *reference counting* and *garbage collection*. Conceptually, the difference between these methods is that reference counting occurs incrementally on individual assignments, while garbage collection occurs as a large batch-oriented task that is run on demand.

Reference Counting This technique augments each object with a counter that tracks the number of outstanding pointers that refer to it. Thus, at an object's initial allocation, its reference count is set to one. Every assignment to a pointer variable involves adjusting two reference counts. The pointer's pre-assignment value is used to decrement the reference count of that object, and its post-assignment value is used to increment the reference count of that object. When an object's reference count drops to zero, the object is added to the free list. (In practice, the system may implement multiple free lists, as described earlier.) When an object is freed, the system must account for the fact that it is discarding any pointers contained in the object. Consider, for example, discarding the last pointer to an abstract syntax tree. Freeing the root node of the tree decrements the reference counts of its children, which decrement the reference counts of their children, and so on, until all of the root's descendants are free.

The presence of pointers inside allocated objects creates three problems for reference counting schemes:

1. the running code needs a mechanism for distinguishing pointers from other data – To distinguish pointers from other data, reference counting systems either store extra information in the header field for each object, or they limit the range of pointers to less than a full word and use the remaining bits to "tag" the pointer.

2. the amount of work done for a single decrement can grow quite large – In systems where external constraints require bounded time for deallocation, the run-time system can adopt a more complex protocol that limits the number of objects deallocated on each pointer assignment. Keeping a queue of objects with reference counts of zero and deallocating a small fixed number on each reference-count adjustment can ensure bounded-time operations, albeit at an increase in the number of instructions required per deallocation.

3. the program can form cyclic graphs with pointers – The reference counts for a cyclic data structure cannot be decremented to zero. When the last external pointer is discarded, the cycle becomes both unreachable and

non-recyclable. To ensure that such objects are freed, the programmer must break the cycle before discarding its last external pointer.[4]

Reference counting incurs additional cost on every pointer assignment. The amount of work done on a specific assignment can be bounded; in any well-designed scheme, the total cost can be limited to some constant factor of 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.

Garbage Collection With these techniques, the allocator does no deallocation until it has run out of free space. At that point, it pauses the program's execution and examines the pool of allocated memory to discover unused objects. When it finds unused objects, it reclaims their space by deallocating them. Some techniques compact memory at the same time; in general, this requires an extra level of indirection on each access. Other methods leave objects in their original locations; this simplifies access at the cost of possible fragmentation of the available memory pool.

Logically, garbage collection proceeds 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 assumes that any object not reachable in this manner 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 by using a marking algorithm. The collector needs a bit for each object in the heap, called a *mark bit*. These can be stored in the object's header, alongside tag information used to record pointer locations or object side. Alternatively, the collector can create a dense bit-map for the heap when needed. The initial step clears all the mark bits and builds a worklist that contains all of the pointers stored in registers and in activation records that correspond to current or pending 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.

Figure 7.13 presents a high-level sketch of a marking algorithm. It is a simple fixed-point computation that halts because the heap is finite and the marks prevent a pointer contained in the heap from entering the `Worklist` more than once. The cost of marking is proportional the number of pointers contained in program variables plus the size of the heap.

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

[4] The alternative—performing reachability analysis on the pointers at run-time—is quite expensive. It negates most of the benefits of reference counting.

```
Clear all marks
Worklist $\leftarrow$ { pointer values from ARs & registers }
while (Worklist $\neq \emptyset$)
   p $\leftarrow$ head of Worklist
   if (p->object is unmarked)
      mark p->object
      add pointers from p->object to Worklist
```

Figure 7.13: A Marking Algorithm

the final line of the `while` loop.

- In a precise collector, the compiler and run-time system know the type of each object, and, hence, its layout. This information can be recorded in object headers, or it can be implicitly known from the type structure of the language. Either way, with precise knowledge, only real pointers are followed in the marking phase.
- In a conservative marking phase, the compiler and run-time system are unsure about the type and layout of some, if not all, objects. Thus, when an object is marked, the system considers each field as a possible pointer. If its value might be a pointer, it is treated as a pointer.[5]

Conservative collectors fail to reclaim some objects that a precise collector would find. However, they have been retrofitted successfully 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. (In a conservative collector, some marked objects may be dead, too. The collector lets them survive because of the uncertainty of its knowledge about object layout.) As the second phase traverses the heap to collect the garbage, it can reset the mark fields to "unmarked." This lets the collector avoid the initial traversal of the heap in the marking phase.

Mark-Sweep Collectors The *mark-sweep* collectors reclaim and recycle objects by making 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 tricks used to coalesce blocks in the first-fit allocator apply. If compaction is desirable, it can be implemented by incrementally shuffling live objects downward during the sweep, or with a post-sweep compaction pass.

[5] For example, 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. Using an indirection table to facilitate compaction can further reduce the range of valid pointers.

Copying Collectors The *copying collectors* divide memory into two pools, an *old* pool and a *new* pool. At any point in time, the allocator operates from the old pool. The simplest *copying collector* is called *stop-and-copy.* When an allocation fails, the stop-and-copy collector copies all the live data from the old pool into the new pool, and swaps the names "old" and "new." The act of copying live data compacts it; after collection, all the free space is in a single contiguous block. This can be done in two passes, like mark-sweep, or it can be done incrementally, as live data is discovered. The incremental scheme can modify the original copy of the in the old pool to avoid copying it more than once.

Comparing the Techniques Implicit deallocation frees the programmer from worrying about when to release memory and from tracking down the inevitable storage leaks that result from attempting to manage allocation and deallocation explicitly. Both mark-sweep and copying collectors have advantages and disadvantages. In practice, the benefits of implicit deallocation outweigh the disadvantages of either scheme for most applications.

The mark-sweep collectors examine the complete pool of memory that can be allocated, while copying collectors only touch 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 cost of various memory references.

Because it moves live objects, a copying collector can easily deallocate a dead cyclic structure; it never gets copied. Mark-sweep collectors have problems discovering that cyclic structures are dead, since they point to themselves.

Copying collectors require either a mechanism for updating stored pointers, or the use of an indirection table for each object access. This added cost per access, however, lets the collector compact memory and avoid fragmentation. Mark-sweep collectors can compact memory, but it requires the addition of an indirection table, just as with the copying collector.

In general, a good implementor can make either mark-sweep or copying work well enough that they are acceptable for most applications. Some applications, such as real-time controllers, will have problems with any unpredictable overhead. These applications should be coded in a fashion that avoids reliance on implicit deallocation.

7.8 Object-oriented Languages

This section will appear as a handout later in the semester.

7.9 Summary and Perspective

The primary rationale for moving beyond assembly language is to provide a more abstract programming model and, thus, raise both programmer productivity and the understandability of programs. Each abstraction added to the programming language requires translation into the ISA of the target machine before it can execute. This chapter has explored the techniques commonly used to translate some of these abstractions—in particular, how the introduction of procedures creates new abstractions for the transfer of control, for naming, and for providing interfaces for use by other procedures and other programmers.

Procedural programming was discovered quite early in the history of programming. Some of the first procedures were debugging routines written for early computers; the availability of these pre-written routines allowed programmers to understand the run-time state of an errant program. Without such routines, tasks that we now take for granted, such as examining the contents of a variable or asking for a trace of the call stack, required the programmer to enter long machine language sequences (without error).

The introduction of lexical scoping in languages like Algol-60 influenced language design for decades. Most modern programming languages carry forward some of the Algol philosophy toward naming and addressability. The early implementors of scoped languages developed clever techniques to keep the price of abstraction low; witness the global display with its uniform cost for accessing names across an arbitrary distance in lexical scope. These techniques are still used today.

Modern languages have added some new twists. By treating procedures as first-class objects, systems like Scheme have created new control-flow paradigms. These require variations on traditional implementation techniques—for example, heap allocation of activation records to support continuations. Similarly, the growing acceptance of implicit deallocation requires occasional conservative treatment of a pointer (as discussed in Chapter 14). If the compiler can exercise a little more care and free the programmer from ever deallocating storage again, that appears to be a good tradeoff.[6]

As new programming paradigms come into vogue, 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 run-time penalty for using higher levels of abstraction.

Questions

1. The compiler writer can optimize the allocation of ARs in several ways. For example, the compiler might:

    (a) Allocate ARs for leaf procedures (those that make no procedure calls) statically.

    (b) Combine the ARs for procedures that are always called together. (When α is called, it always calls β.)

    (c) Use an arena-style allocator in place of heap allocation of ARs.

    For each scheme, consider the following questions:

[6] Generations of experience suggest that programmers are not effective at freeing all the storage that they allocate. This is precisely the kind of detail that computers should be used to track!

   (a) What fraction of the calls might benefit? In the best case? In the worst case?

   (b) What is the impact on run-time space utilization?

2. What is the relationship between the notion of a linkage convention and the construction of large programs? of inter-language programs? How can the linkage convention provide for an inter-language call?

Chapter 8

Code Shape

8.1 Introduction

One of the compiler's primary tasks is to emit code that faithfully implements the various source-language constructs used in the input program. In practice, some of these constructs have many different implementations on a specific target-machine—variations that produce the same results using different operations or different techniques. Some of these implementations will be faster than others; some will use less memory; some will use fewer registers; some might consume less power during execution. The various implementations are equivalent, in that they produce the same answers. They differ in layout, in cost, in choice of instructions to implement various source-language constructs, and in the mapping of storage locations to program values. We consider all of these issues to be matters of n*code shape.*

Code shape has a strong impact on the behavior of code generated by a compiler, and on the ability of the optimizer and the back end to improve it. Consider, for example, the way that a C compiler might implement a case statement that switched on a single-byte character value. The compiler might implement the the `switch` statement with a cascaded series of `if-then-else` statements. Depending on the layout, this could produce quite different results. If the first test is for zero, the second for one, and so on, the cost devolves to linear search over a field of two-hundred fifty-six keys. If characters are uniformly distributed, the average case would involve one hundred twenty-eight tests and branches—an expensive way to implement a case statement. If the tests perform a binary search, the average case would involve eight tests and branches—a more palatable number. If the compiler is willing to spend some data space, it can construct a table of two hundred fifty-six labels, and interpret the character by loading the corresponding table entry and branching to it—with a constant overhead per character.

All of these are legal implementations of the case statement. Deciding which implementation makes sense for a particular instance of the case statement depends on many factors. In particular, the number of individual cases and the

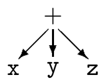
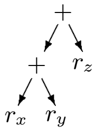
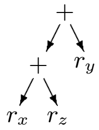
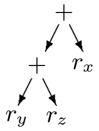
	Source code	*Low-level, three address code*		
Code	`x + y + z`	$r_x + r_y \rightarrow r_1$ $r_1 + r_z \rightarrow r_2$	$r_x + r_z \rightarrow r_1$ $r_1 + r_y \rightarrow r_2$	$r_y + r_z \rightarrow r_1$ $r_1 + r_x \rightarrow r_2$
Tree				

Figure 8.1: Alternate Code Shapes for `x + y + z`

relative frequency of execution are important, as is detailed knowledge of the cost structure for branching on the target machine. Even when the compiler cannot determine the information that it needs to make the best choice, it must make a choice. The difference between the possible implementations, and the compiler's choice, are matters of code shape.

As another example, consider the simple expression `x+y+z`. Figure 8.1 shows several ways of implementing the 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 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. (We assume that each variable is in an appropriately-named register.) Commutativity makes all three orders legal; the compiler must choose between them.

Left associativity would produce the first binary tree. This tree seems "natural," in that left associativity corresponds to our left-to-right reading style. If, however, we replace `y` with the literal constant `2` and `z` with `3`, then we discover that this shape for the expression hides a simple optimization. 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 the left associative form, however, `2 + 3` never occurs. The right associative form, of course, exposes this optimization. Each prospective tree, however, has an assignment of variables and constants to `x`, `y`, and `z` that makes it look bad.

As with the case statement, the best shape for this expression cannot be known without understanding information that may not be contained in the statement itself. Similar, but less obvious effects occur. If, for example, the expression `x + y` has been computed recently and neither the value of `x` nor the value of `y` has changed, then using the center form would let the compiler replace the first operation $r_x + r_y \rightarrow r_1$ with a reference to the previously computed value. In this situation, the best choice between the three evaluation orders might depend on context from the surrounding code.

This chapter explores the code shape issues that arise in generating code for 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 subsequent chapters.

8.2 Assigning Storage Locations

A procedure computes many values. Some of these have names in the source code; in an Algol-like language, the programmer provides a name for each variable. Other values have no explicit names; for example, the value `i-3` in the expression `A[i-3,j+2]` has no name. Named values are exposed to other procedures and to the debugger. They have defined lifetimes. These facts limit where the compiler can place them, and how long it must preserve them. For unnamed values, such as `i-3`, the compiler must treat them in a way consistent with the meaning of the program. This leaves the compiler substantial freedom in determining where these values reside and how long they are retained.

The compiler's decisions about both named and unnamed values have a strong impact on the final code that it produces. In particular, decisions about unnamed values determine the set of values exposed to analysis and transformation in the optimizer. In choosing a storage location for each value, the compiler must observe the rules of both the source language and the target machine's memory hierarchy. In general, it can place a value in a register or in memory. The memory address space available to the program may be divided into many distinct subregions, or *data areas*, as we saw in Figure 7.11.

Algol-like languages have a limited number of data-areas, defined by the source language's name scoping rules (see Section 7.3). Typically, each procedure has a data area for its local scope; it may also have a procedure-related static data area. Global variables can be treated as residing in either a single global data area, or in a distinct data area for each global variable. Some languages add other scopes. For example, C has static storage that is accessible to every procedure in a given file, but no procedure-related static storage. It also adds a lexical scope for individual "blocks", any code region enclosed in curly braces.

Object-oriented languages have a richer set of data areas. They are, quite naturally, organized around the name space of objects. Each object has its own local data area, used to hold object-specific values—sometimes called *instance variables*. Since classes are objects, they have a data area; typically some of the values in the data area of a class are accessible to each method defined by the class. The language may provide a mechanism for the method to define local variables; this scope creates a data-area equivalent to the procedure local data area of an Algol-like language. Objects themselves can be in the global scope; alternatively, they can be declared as instance variables of some other object.

Any particular code fragment has a structured view of this sea of data areas. It can access data local to the method that contains it, instance variables of the object named as `self`, some instance variables of its class, and, depending on inheritance, some instance variables of its superclasses. This inheritance of data areas differs from the notion of accessibility provided by lexical scoping in an Algol-like language. It arises from the inheritance relations among data objects, rather than any property of the program text.

Laying Out Data Areas To assign variables in an Algol-like language to storage classes, the compiler might apply rules similar to these:

1. x is declared locally, and
   (a) its value is not preserved across invocations ⇒ procedure-local storage
   (b) its value is preserved across invocations ⇒ procedure-static storage
2. x is declared globally ⇒ global storage
3. x is allocated under program control ⇒ the run-time heap

The different storage locations have different access costs. Procedure-local storage can reside in the procedure's AR. Since the procedure always has a pointer to its AR, these values can be accessed directly with operations like ILOC's `loadAO` and `storeAO` (or their immediate forms `loadAI` and `storeAI`). In addition, because the typical procedure references its AR for parameters, for register-spill locations, and for local variables, the AR is likely to remain in the processor's primary cache. Access to local variables of other procedures is more complex; Section 7.5 detailed two mechanisms for accomplishing this task: access links and a display.

Accessing static or global data areas may require additional work to establish addressability. Typically, this requires a `loadI` to get the run-time address of some relocatable symbol (an assembly-language label) into a register where it can be used as a base address. If the procedure repeatedly refers to values in the same data area, the base address may end up residing in a register. To simplify address calculations, many compilers give each global variable a unique label. This eliminates an addition by the variable's offset; in ILOC, that addition comes without cost in a `loadAO` or `loadAI` operation.

Keeping a Value in a Register In addition to assigning a storage class and location to each value, the compiler must determine whether or not it can safely keep the value in a register. If the value can safely reside in a register, and the register allocator is able to keep the value in a register for its entire lifetime, it may not need space in memory. A common strategy followed by many modern compilers is to assign a fictional, or virtual, register to each value that can legally reside in a register, and to rely on the register allocator to pare this set down to a manageable number. In this scheme, the compiler either assigns a virtual register or a memory address to each value, but not both. When the register allocator decides that some virtual register must be converted into a memory reference, the allocator assigns it space in memory. It then inserts the appropriate loads and stores to move the value between a register and its home in memory.

To determine whether or not a value can be kept in a register, the compiler tries to determine the number of distinct names by which the code can access a given value. For example, a local variable can be kept in a register as long as its address is never taken and it is not passed as a call-by-reference parameter to another procedure. Either of these actions creates a second path for accessing the variable. Consider the following fragment in C:

```
void fee();
{
  int a, *b;
    ...
  b = &a;
    ...
}
```

The assignment of &a to b creates a second way for subsequent statements to access the contents of a. Any reference to *b will return the contents of a. The compiler cannot safely keep a in a register, unless it performs enough analysis to prove that *b is never referenced while b has the value &a. This involves examining every statement in the elided portion of the code. This may include other pointer assignments, addressing operations, and indirect access. These can make the analysis difficult.

For example, if we add *b = a++; after the assignment to b, what is the value of a after the statement executes? Both sides of the new assignment refer to the same location. Is the autoincrement to a performed before the store through b, or vice-versa? If fee contains any conditionally executed paths, then b can receive different values along different paths through the procedure. This would require the compiler to prove small theorems about the different values that can reach each assignment before deciding whether or not keeping a in a register is safe. Rather than prove such theorems, the typical C compiler will assign a to a memory location instead of a register.

A value that can be kept in a register is sometimes called an *unambiguous value*; a value that can have more than one name is called an *ambiguous value*. Ambiguity arises in several ways. Pointer-based variables are often ambiguous; interactions between call-by-reference formal parameters and name scoping rules can create ambiguity as well. Chapter 13 describes the analysis required to shrink the set of ambiguous values.

Since treating an ambiguous value as an unambiguous value can cause incorrect behavior, the compiler must treat any value as ambiguous unless it can prove that the value is unambiguous. Ambiguous values are kept in memory rather than in registers; they are loaded and stored as necessary.[1] Careful reasoning about the language can help the compiler. For example, in C, any local variable whose address is never taken is unambiguous. Similarly, the ANSI C standard requires that references through pointer variables be type consistent;

[1] The compiler could, in fact, keep an ambiguous value in a register over a series of statements where no other ambiguous value is referenced. In practice, compilers simply relegate ambiguous values to memory, rather than institute the kind of statement-by-statement tracking of values necessary to discover a region where this would be safe.

thus an assignment to `*b` can only change the value of a location for which `b` is a legal pointer. (The ANSI C standard exempts character pointers from this restriction. Thus, an assignment to a character pointer can change a value of any type.) The analysis is sufficiently difficult, and the potential benefits large enough, that the ANSI C standard has added the `restrict` keyword to allow the programmer to declare that a pointer is unambiguous.

Machine Idiosyncrasies Within a storage class, some machine-specific rules may apply. The target machine may restrict the set of registers where a value can reside. These rules can take many forms.

- Some architectures have required double-precision, floating-point values to occupy two adjacent registers; others limit the choices to pairs beginning with specific registers, such as the even-numbered registers.

- Some architectures partition the register set into distinct sets of registers, or *register classes*. Sometimes these are disjoint, as is commonly the case with "floating-point" and "general purpose" registers. Sometimes these classes overlap, as is often the case with "floating point" and "double precision" registers. Other common register classes include condition code registers, predicate registers, and branch target registers.

- Some architectures partition the register set into multiple disjoint register files and group functional units around them; each functional unit has fast access to the registers in its associated set and limited access to registers in the other register sets. This allows the architect to add more functional units; it requires that compiler pay attention to the placement of both operations an data.

The compiler must handle all of these target-specific rules.

Target-specific issues arise with memory resident values, as well. Many architectures restrict the starting address of a value based on its perceived data type. Thus, integer and single-precision floating-point data might be required to start on a word boundary (an address that is an integral multiple of the word size), while character data might begin at any even address. Other restrictions require alignment to multi-word boundaries, like double-word or quad-word boundaries.

The details of storage assignment can directly affect performance. As memory hierarchies become deeper and more complex, issues like spatial locality and reuse have a large effect on running time. Chapter 14 gives an overview of the techniques developed to address this aspect of performance. Most of the work operates as a post-pass to storage assignment, correcting problems rather than predicting them before relative addresses are assigned.

8.3 Arithmetic Expressions

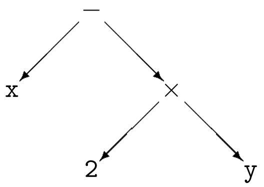
Modern processors provide broad support for arithmetic operations. A typical RISC-machine has a full complement of three-address operations, including addition, subtraction, multiplication, division, left and right shifts, and boolean

```
expr(node) {
  int result, t1, t2;
  switch(type(node))
  {
    case $\times, \div, +, -$:
      t1 ← expr(left child(node));
      t2 ← expr(right child(node));
      result ← NextRegister();
      emit(op(node), t1, t2, result);
      break;

    case IDENTIFIER:
      t1 ← base(node);
      t2 ← offset(node);
      result ← NextRegister();
      emit(loadAO, t1, t2, result);
      break;

    case NUMBER:
      result ← NextRegister();
      emit(loadI, val(node), none,
           result);
      break;
  }
  return result;
}
```

Treewalk Code Generator



Expression Tree for
x − 2 × y

```
loadI   @x        ⇒ r1
loadAO  r0,r1     ⇒ r2
loadI   4         ⇒ r3
loadI   @y        ⇒ r4
loadAO  rarp, r4  ⇒ r5
mult    r3, r5    ⇒ r6
sub     r2, r6    ⇒ r7
```

Naive Code

Figure 8.2: Simple Treewalk for Expressions

operations. The three-address structure of the architecture provides the compiler with the opportunity to create an explicit name for the result of any operation; this lets the compiler select a name space that preserves values which may be re-used later in the computation. It also eliminates one of the major complications of two-address instructions—deciding which operand to destroy in each executed operation.

To generate code for a trivial expression, like x + y, the compiler first emits code to ensure that the value of x and y are in known registers, say $\mathtt{r}_x$ and $\mathtt{r}_y$. If x is stored in memory, at some offset "@x" in the current activation record the resulting code might be

```
loadI   @x         ⇒ r1
loadAO  rarp, r1   ⇒ rx
```

If, however, the value of x is already in a register, the compiler can simply use that register's name in place of $\mathtt{r}_x$. The compiler follows a similar chain of decisions to ensure that y is in a register. Finally, it emits an instruction to perform the addition, such as

```
add   r_x, r_y   ⇒ r_t
```

If the expression is a syntax tree, this scheme fits naturally into a postorder walk of the tree. The code in Figure 8.2 does this by embedding the code-generating actions into a recursive treewalk routine. Notice that the same code handles $+$, $-$, $\times$, and $\div$. From a code-generation perspective, the binary operators are interchangeable (ignoring commutativity). Applying the routine from Figure 8.2 to the expression x $-$ 2 $\times$ y, produces the results shown at the bottom of the figure. This assumes that neither x nor y is already in a register.

Many issues affect the quality of the generated code. For example, the choice of storage locations has a direct impact, even for this simple expression. If y were in a global data area, the sequence of instructions needed to get y into a register might require an additional loadI to obtain the base address, and a register to hold that value. Alternatively, if y were in a register, the two instructions used to load it into r_5 would be omitted and the compiler would use the name of the register holding y directly in the mult instruction. Keeping the value in a register avoids both the memory access and any supporting address calculation. If both x and y were in registers, the seven instruction sequence would be shortened to a three instruction sequence (two if the target machine supports an immediate multiply instruction).

Code shape decisions encoded into the treewalk code generator have an effect, too. The naive code in the figure uses seven registers (plus r_{arp}). It is tempting to assume that the register allocator can reduce the number of registers to a minimum. For example, the register allocator could rewrite the expression as:

```
loadI    @x          ⇒ r1
loadAO   r_arp, r1   ⇒ r1
loadI    2           ⇒ r2
loadI    @y          ⇒ r3
loadAO   r_arp, r3   ⇒ r3
mult     r2, r3      ⇒ r2
sub      r1, r2      ⇒ r2
```

This drops register use from seven registers to three (excluding r_{arp}). (It leaves the result in r_2 so that both x, in r_1, and y, in r_3, are available for later use.)

However, loading x before computing 2 $\times$ y still wastes a register—an artifact of the decision in the treewalk code generator to evaluate the left child before the right child. Using the opposite order would produce the following code sequence:

```
loadI    @y          ⇒ r_1
loadAO   r_arp, r_1  ⇒ r_2
loadI    2           ⇒ r_3
mult     r_3, r_2    ⇒ r_4
loadI    @x          ⇒ r_5
loadAO   r_arp,r_5   ⇒ r_6
sub      r_6, r_4    ⇒ r_7
```

The register allocator could rewrite this to use only two registers (plus $\mathtt{r}_{arp}$):

```
loadI    @y          ⇒ r_1
loadAO   r_arp, r_1  ⇒ r_1
loadI    2           ⇒ r_2
mult     r_2, r_1    ⇒ r_1
loadI    @x          ⇒ r_2
loadAO   r_arp,r_2   ⇒ r_2
sub      r_2, r_1    ⇒ r_1
```

The allocator cannot fit all of x, y, and x + 2 × y into two registers. As written, the code preserves x and not y.

Of course, evaluating the right child first is not a general solution. For the expression 2 × y + x, the appropriate rule is "left child first." Some expressions, such as x + (5 + y) × 7 defy a static rule. The best evaluation order for limiting register use is 5 + y, then × 7, and finally + x. This requires alternating between right and left children.

To choose the correct evaluation order for subtrees of an expression tree, the compiler needs information about the details of each subtree. To minimize register use, the compiler should first evaluate the more demanding subtree—the subtree that needs the most registers. The code must must preserve the value computed first across the evaluation of the second subtree; thus, handling the less demanding subtree first increases the demand for registers in the more demanding subtree by one register. Of course, determining which subtree needs more registers requires a second pass over the code.

This set of observations leads to the Sethi-Ullman labeling algorithm (see Section 9.1.2). They also make explicit the idea that taking a second pass over an expression tree can lead to better code than the compiler can generate in a single pass [31]. This should not be surprising; the idea is the basis for multi-pass compilers. An obvious corollary suggests that the second and subsequent passes should know how large to make data structures such as the symbol table.

Accessing Parameter Values The treewalk code generator implicitly assumes that a single access method works for all identifiers. Names that represent formal parameters may need different treatment. A call-by-value parameter passed in the AR can be handled as if it were a local variable. A call-by-reference parameter passed in the AR requires one additional indirection. Thus, for the call-by-reference parameter x, the compiler might generate

Digression: Generating `loadAI` *instructions*

A careful reader might notice that the code in Figure 8.2 never generates ILOC's `loadAI` instruction. In particular, it generates the sequence

```
loadI   @x        ⇒ r1
loadAO  r_arp,r1  ⇒ r2
```

when the operation `loadAI` r_{arp},`@x` ⇒ r_2 achieves the same effect with one fewer register and one fewer instruction.

Throughout the book, we have assumed that it is preferable to generate this two operation sequence rather than the single operation. Two reasons dictate this choice:

1. The longer code sequence gives a register name to the label `@x`. If the label is reused in contexts other than a `loadAO` instruction, having an explicit name is useful. Since `@x` is typically a small integer, this may occur more often than you would expect.

2. The two instruction sequence leads to a clean functional decomposition in the code generator, as seen in Figure 8.2. Here, the code uses two routines, *base* and *offset*, to hide the details of addressability. This interface lets *base* and *offset* hide any data structures that they use.

Subsequent optimization can easily convert the two instruction sequence into a single `loadAI` if the constant offset is not reused. For example, a graph-coloring register allocator that implements *rematerialization* will do this conversion if the intermediate register is needed for some other value.

If the compiler needs to to generate the `loadAO` directly, two approaches make sense. The compiler writer can pull the case logic contained in *base* and *offset* up into the case for *IDENTIFIER* in Figure 8.2. This accomplishes the objective, at the cost of less clean and modular code. Alternatively, the compiler writer can have *emit* maintain a small instruction buffer and perform peephole-style optimization on instructions as they are generated (see Section 15.2). Keeping the buffer small makes this practical. If the compiler follows the "more demanding subtree first" rule, the offset will be generated immediately before the `loadAO` instruction. Recognizing a `loadI` that feeds into a `loadAO` is easy in the peephole paradigm.

```
loadI   @x        ⇒ r1
loadAO  r_arp,r1  ⇒ r2
load    r2        ⇒ r3
```

to obtain `x`'s value. The first two operations move the memory address of the parameter's value into r_2. The final operation moves that value into r_3.

Many linkage conventions pass the first few parameters in registers. As written, the code in Figure 8.2 cannot handle a value that is permanently kept in a register. The necessary extension, however, is simple.

For call-by-value parameters, the *IDENTIFIER* case must check if the value is already in a register. If so, it assigns the register number to *result*. Otherwise, it uses the code to load the value from memory. It is always safe to keep call-by-value parameters in a register in the procedure where they are declared.

For a call-by-reference parameter that is passed in a register, the compiler only needs to emit the single operation that loads the value from memory. The value, however, must reside in memory across each statement boundary, unless the compiler can prove that it is unambiguous—that is, not aliased. This can require substantial analysis.

Function Calls in an Expression So far, we have assumed that the basic operands in an expression are variables and temporary values produced by other subexpressions. Function calls also occur as references in expressions. To evaluate a function call, the compiler simply generates the calling sequence needed to invoke the function (see Section 7.6 and 8.9) and emits the code necessary to move the returned value into a register. The procedure linkage limits the impact on the calling procedure of executing the function.

The presence of a function call may restrict the compiler's ability to change the expression's evaluation order. The function may have side effects that modify the value of variables used in the expression. In that situation, the compiler must adhere strictly to the source language's evaluation order; without side effects, the compiler has the freedom to select an evaluation order that produces better code. Without knowledge about the possible side effects of the call, the compiler must assume the worst case—that the function call results in a modification to every variable that the function could possible touch. The desire to improve on "worst case" assumptions such as this motivated much of the early work in interprocedural data-flow analysis (see Section 13.4).

Other Arithmetic Operations To handle additional arithmetic operations, we can extend our simple model. The basic scheme remains the same: get the operands into registers, perform the operation, and store the result if necessary. The precedence encoded in the expression grammar ensures the intended ordering. Unary operators, such as unary minus or an explicit pointer dereference, evaluate their sole subtree and then perform the specified operation. Some operators require complex code sequences for their implementation (*i.e.,* exponentiation, trigonometric functions, and reduction operators). These may be expanded directly inline, or they may be handled with a function call to a library supplied by the compiler or the operating system.

Mixed-type Expressions One complication allowed by many programming languages is an operation where the operands have different types. (Here, we are concerned primarily with base types in the source language, rather than programmer-defined types.) Consider an expression that multiplies a floating-point number by an integer. First, and foremost, the source language must define the meaning of such a mixed-type expression. A typical rule converts both operands to the more general type, performs the operation in the more general type, and produces its result in the more general type. Some machines provide instructions to directly perform these conversions; others expect the compiler to generate complex, machine-dependent code. The operation that consumes the result value may need to convert it to another type.

The notion of "more general" type is specified by a conversion table. For example, the Fortran 77 standard specifies the following conversions for addition:

+	*Integer*	*Real*	*Double*	*Complex*
Integer	INTEGER	REAL	DOUBLE	COMPLEX
Real	REAL	REAL	DOUBLE	COMPLEX
Double	DOUBLE	DOUBLE	DOUBLE	ILLEGAL
Complex	COMPLEX	COMPLEX	ILLEGAL	COMPLEX

The standard further specifies a formula for each conversion. For example, to convert INTEGER to COMPLEX, the compiler converts the INTEGER to a REAL, and uses the REAL value as the real part of the complex number. The imaginary part of the complex number is set to zero.

Most conversion tables are symmetric. Occasionally, one is asymmetric. For example, PL/I had two different representations for integers: a straightforward binary number, denoted fixed binary, and a binary-coded decimal (or BCD), denoted fixed decimal. In the conversion table for addition, the result type of adding a fixed decimal and a fixed binary depended on the order of the arguments. The resulting operation had the type of the first argument.

For user-defined types, the compiler will not have a conversion table that defines each specific case. However, the source language still defines the meaning of the expression. The compiler's task is to implement that meaning; if conversion is illegal, then it should be prevented. As seen in Chapter 5, illegal conversions can sometimes be detected at compile time. In such circumstances, the compiler should report the possible illegal conversion. When such a compile-time check is either impossible or inconclusive, the compiler must generate run-time checks to test for the illegal cases. If the test discovers an illegal conversion, it should raise a run-time error.

The IBM PL/I compilers include a feature that let the programmer avoid all conversions. The `unspec` function converted any value, including the left-hand side of an assignment statement, to a bit string. Thus, the programmer could assign a floating-point number to an appropriately-sized character string. In essence, `unspec` was a short cut around the entire type system.

Assignment as an Operator Most Algol-like languages implement assignment with the following simple rules.

1. Evaluate the right hand side of the assignment to a value.
2. Evaluate the left hand side of the assignment to an address.
3. Move the value into the location specified by the left hand side.

Thus, in a statement like `x` ← `y`, the two expressions `x` and `y` are evaluated differently. Since `y` appears to the right of the assignment, it is evaluated to a value. Since `x` is to the left of the assignment, it is evaluated to an address. The right and left sides of an assignment are sometimes referred to as an *rvalue* and an *lvalue*, respectively, to distinguish between these two modes of evaluation.

An assignment can be a mixed-type expression. If the rvalue and lvalue have different types, conversion may be required. The typical source-language rule has the compiler evaluate the *rvalue* to its natural type—the type it would generate without the added context of the assignment operator. That result is then converted to the type of the lvalue, and stored in the appropriate location.

Commutativity, Associativity, and Number Systems Sometimes, the compiler can take advantage of algebraic properties of the various operators. For example, addition, multiplication, and exclusive or are all commutative. Thus, if the compiler sees a code fragment that computes `x + y` and then computes `y + x`, with no intervening assignments to either `x` or `y`, it should recognize that they compute the same value. Similarly, if it sees the expressions `x + y + z` and `w + x + y`, it should consider the fact that `x + y` is a common subexpression between them. If it evaluates both expressions in a strict left-to-right order, it will never recognize the common subexpression, since it will compute the second expression as `w + x` and then (w + x) + y.

The compiler should consider commutativity and associativity as are discussed in Chapter 14. Reordering expressions can lead to improved code. However, a brief warning is in order.

> *Floating-point numbers on computers are not real numbers, in the mathematical sense. They approximate a subset of the real numbers, but the approximation does not preserve associativity. As a result, compilers should not reorder floating-point computations.*

We can assign values to `x`, `y`, and `z` such that (in floating-point arithmetic) $z - x = z$, $z - y = z$, but $z - (x + y) \neq z$. In that case, reordering the computation changes the numerical result. By adding the smaller values, `x` and `y`, first, the computation maximizes the retained precision. Reordering the computation to compute one of the other possible partial sums would throw away precision. In many numerical calculations, this could change the results. The code might execute faster, but produce incorrect results.

This problem arises from the approximate nature of floating-point numbers; the mantissa is small relative to the range of the exponent. To add two numbers, the hardware must normalize them; if the difference in exponents is larger than the base ten precision of the mantissa, the smaller number will be truncated to zero. The compiler cannot easily work its way around the issue. Thus, it should obey the cardinal rule and not reorder floating-point computations.

8.4 Boolean and Relational Values

Most programming languages operate on a richer set of values than numbers. Usually, this include *boolean*, or logical, values and *relational*, or comparison, values. Programs use boolean and relational expressions to control the flow of

$$\begin{array}{lcl}
\textit{expr} & \rightarrow & \neg\ \textit{or-term} \\
 & | & \textit{or-term} \\
\textit{or-term} & \rightarrow & \textit{or-term} \vee \textit{and-term} \\
 & | & \textit{and-term} \\
\textit{and-term} & \rightarrow & \textit{and-term} \wedge \textit{bool} \\
 & | & \textit{bool} \\
\textit{bool} & \rightarrow & \textit{r-expr} \\
 & | & \texttt{true} \\
 & | & \texttt{false} \\
\textit{r-expr} & \rightarrow & \textit{r-expr} < \textit{n-expr} \\
 & | & \textit{r-expr} \leq \textit{n-expr} \\
 & | & \textit{r-expr} = \textit{n-expr} \\
 & | & \textit{r-expr} \neq \textit{n-expr} \\
 & | & \textit{r-expr} \geq \textit{n-expr} \\
 & | & \textit{r-expr} > \textit{n-expr} \\
 & | & \textit{n-expr} \\
\textit{n-expr} & \rightarrow & \textit{n-expr} + \textit{term} \\
 & | & \textit{n-expr} - \textit{term} \\
 & | & \textit{term} \\
\textit{term} & \rightarrow & \textit{term} \times \textit{factor} \\
 & | & \textit{term} \div \textit{factor} \\
 & | & \textit{factor} \\
\textit{factor} & \rightarrow & (\ \textit{expr}\ ) \\
 & | & \texttt{number} \\
 & | & \texttt{identifier}
\end{array}$$

Figure 8.3: Adding booleans and relationals to the expression grammar

execution. Much of the power of modern programming languages derives from the ability to compute and test such values.

To express these values, language designers add productions to the standard expression grammar, as shown in Figure 8.3. (We have used the symbols ¬ for `not`, ∧ for `and`, and ∨ for `or` to avoid any confusion with the corresponding ILOC operations.) The compiler writer must, in turn, decide how to represent these values and how to compute them. With arithmetic expressions, the design decisions are largely dictated by the target architecture, which provides number formats and instructions to perform basic arithmetic. Fortunately, processor architects appear to have reached a widespread agreement about how to support arithmetic. The situation is similar for boolean values. Most processors provide a reasonably rich set of boolean operations. Unfortunately, the handling of relational expressions varies from processor to processor. Because the relational operators in programming languages produce, at least conceptually, boolean results, the issues of representation and code generation for relationals and booleans are closely related.

8.4.1 Representations

Traditionally, two representations have been proposed for boolean and relational values: a numerical representation and a positional encoding. The former assigns numerical values to `true` and `false`, then uses the arithmetic and logical instructions provided on the target machine. The latter approach encodes the value of the expression as a position in the executable code. It uses the hardware comparator and conditional branches to evaluate the expression; the different control-flow paths represent the result of evaluation. Each approach has examples where it works well.

Numerical Representation When a boolean or relational value is stored into a variable, the compiler must ensure that the value has a concrete representation. To accomplish this, the compiler assigns numerical values to `true` and `false` so that hardware instructions such as `and`, `or`, and `not` will work. Typical values are zero for `false` and either one or negative one for `true`. (In two's complement arithmetic, negative one is a word of all ones.) With this representation, the compiler can use hardware instructions directly for boolean operations.

For example, if `b`, `c`, and `d` are all in registers, the compiler might produce the following code for the expression `b` $\vee$ `c` $\wedge$ $\neg$`d`:

```
not   r_d      ⇒ r_1
and   r_c,r_1  ⇒ r_2
or    r_b,r_2  ⇒ r_3
```

For a comparison, like `x < y`, the compiler must generate code that compares the two operations and then assigns the appropriate value to the result. If the target machine supports a comparison operation that returns a boolean, the code is trivial:

```
cmp_LT   r_x, r_y   ⇒ r_1
```

If, on the other hand, the comparison sets a condition code register that must be read with a conditional branch, the resulting code is longer and more involved.

ILOC is deliberately ambiguous on this point. It includes a comparison operator (`comp`) and a corresponding set of branches that communicate through one of a set of condition code registers, $\texttt{cc}_i$ (see Appendix A). Using this encoding leads to a messier implementation for `x < y`:

```
      comp     r_a, r_b   ⇒ cc_1
      cbr_LT   cc_1       → L_1,L_2
L_1:  loadI    true       ⇒ r_2
      br                  → L_3
L_2:  loadI    false      ⇒ r_2
L_3:  nop
```

This code uses more operations, including branches that are difficult to predict. As branch latencies grow, these branches will become even less desirable.

If the result of `x < y` is used only to determine control flow, an optimization is possible. The compiler need not create an explicit instantiation of the value. For example, a naive translation of the code fragment:

```
if (x < y)
   then statement_1
   else statement_2
```

would produce the following code:

```
       comp     $r_x$, $r_y$   ⇒ $cc_1$        | evaluate x < y
       cbr_LT   $cc_1$         → $L_1$,$L_2$   |
$L_1$: loadI    true           ⇒ $r_2$         | result is true
       br                      → $L_3$         |
$L_2$: loadI    false          ⇒ $r_2$         | result is false
       br                      → $L_3$         |
$L_3$: comp     $r_2$,true     ⇒ $cc_2$        | move $r_2$ into $cc_2$
       cbr_EQ   $cc_2$         → $L_4$,$L_5$   | branch on $cc_2$
$L_4$: code for $statement_1$                  |
       br                      → $L_6$         |
$L_5$: code for $statement_2$                  |
       br                      → $L_6$         |
$L_6$: nop                                     | next statement
```

Explicitly representing `x < y` with a number makes this inefficient.

This sequence can be cleaned up. The compiler should combine the conditional branches used to evaluate `x < y` with the corresponding branches that select either *statement*$_1$ or *statement*$_2$. This avoids executing redundant branches. It eliminates the need to instantiate a value (`true` or `false`) as the result of evaluating `x < y`. With a little thought, the compiler writer can ensure that the compiler generates code similar to this:

```
       comp     $r_x$, $r_y$   ⇒ $cc_1$        | evaluate x < y
       cbr_LT   $cc_1$         → $L_1$,$L_2$   | and branch ...
$L_1$: code for $statement_1$                  |
       br                      → $L_6$         |
$L_2$: code for $statement_2$                  |
       br                      → $L_6$         |
$L_6$: nop                                     | next statement
```

Here, the overhead of evaluating `x < y` has been folded into the overhead for selecting between *statement*$_1$ and *statement*$_2$. Notice that the result of `x < y` has no explicit value; its value is recorded implicitly—essentially in the processor's program counter as it executes either the statement at L_1 or L_2.

Positional Encoding The previous example encodes the expression's value as a position in the program. We call this representation a *positional encoding.* To see the strengths of positional encoding, consider the code required to evaluate the expression `a<b or c<d and e<f`. A naive code generator might emit:

```
       comp    r_a, r_b  ⇒ cc_1
       cbr_LT  cc_1      → L_3,L_1
L_1:   comp    r_c, r_d  ⇒ cc_2
       cbr_LT  cc_2      → L_2,L_4
L_2:   comp    r_e, r_f  ⇒ cc_3
       cbr_LT  cc_3      → L_3,L_4
L_3:   loadI   true      ⇒ r_1
       br                → L_5
L_4:   loadI   false     ⇒ r_1
       br                → L_5
L_5:   nop
```

Notice that this code only evaluates as much of the expression as is required to determine the final value.

With some instruction sets, positional encoding of relational expressions makes sense. Essentially, it is an optimization that avoids assigning actual values to the expression until an assignment is required, or until a boolean operation is performed on the result of the expression. Positional encoding represents the expression's value implicitly in the control-flow path taken through the code. This allows the code to avoid some instructions. It provides a natural framework for improving the evaluation of some boolean expressions through a technique called *short circuit evaluation.* On an architecture where the result of a comparison is more complex than a boolean value, positional encoding can seem quite natural.

The compiler can encode booleans in the same way. A control-flow construct that depends on the controlling expression $(\texttt{w} < \texttt{x} \land \texttt{y} < \texttt{z})$ might be implemented entirely with a positional encoding, to avoid creating boolean values that represent the results of the individual comparisons. This observation leads to the notion of short-circuit evaluation for a boolean expression—evaluating only as much of the expression as is required to determine its value. Short circuiting relies on two boolean identities:

$$\forall\ \texttt{x},\ \texttt{false} \land \texttt{x} = \texttt{false}$$
$$\forall\ \texttt{x},\ \ \texttt{true} \lor \texttt{x} = \texttt{true}$$

Some programming languages, like C, require the compiler to generate code for short-circuit evaluation. For example, the C expression

```
(x != 0 && y/x > 0.001)
```

relies on short-circuit evaluation for safety. If `x` is zero, `y/x` is not defined. Clearly, the programmer intends to avoid the hardware exception triggered for division by zero. The language definition specifies that this code will never perform the division if `x` has the value zero.

The real issue is implicit versus explicit representation. Positional encoding of an $\land$ operation, for example, only makes sense when both of the operands are positionally encoded. If either operand is represented by a numerical value, using the hardware $\land$ operation makes more sense. Thus, positional encoding occurs most often when evaluating an expression whose arguments are produced by other operations (relationals) and whose result is not stored.

8.4.2 Hardware Support for Relational Expressions

A number of specific, low-level details in the instruction set of the target machine strongly influence the choice of a representation for relational values. In particular, the compiler writer must pay attention to the handling of condition codes, compare operations, and conditional move operations, as they have a major impact on the relative costs of the various representations. We will consider four different instruction-level schemes for supporting relational comparisons. Each scheme is an idealized version of a real implementation.

Straight Condition Codes In this scheme, the comparison operation sets a condition code register. The only instruction that interprets the condition code is a conditional branch, with variants that branch on each of the six relations ($<$, $\leq$, $=$, $\geq$, $>$, and $\neq$).

This model forces the compiler to use conditional branches for evaluating relational expressions. If the result is used in a boolean operation or is preserved in a variable, the code converts it into a numerical representation of a boolean. If the only use of the result is to determine control flow, the conditional branch that "reads" the condition code can usually implement the source-level control-flow construct, as well. Either way, the code has at least one conditional branch per relational operator.

The strength of condition-codes comes from another feature that processors usually implement alongside the condition codes. Typically, these processors have arithmetic operations that set the condition code bits to reflect their computed results. If the compiler can arrange to have the arithmetic operations, which must be performed, set the condition code appropriately, then the comparison operation can be omitted. Thus, advocates of this architectural style argue that it allows a more efficient encoding of the program—the code may execute fewer instructions than it would with a comparator that returned a boolean value to a general purpose register.

Conditional Move This scheme adds a conditional move instruction to the straight condition code model. In ILOC, we write conditional move as

$$\texttt{i2i\_<}\quad \texttt{cc}_i\texttt{,r}_j\texttt{,r}_k \quad \Rightarrow \texttt{r}_l$$

If the condition code $\texttt{cc}_i$ matches $<$, then the value of $\texttt{r}_j$ is copied to $\texttt{r}_l$. Otherwise, the value of $\texttt{r}_k$ is copied to $\texttt{r}_l$.

Conditional move retains the potential advantage of the condition code scheme—avoiding the actual comparison operation—while providing a single instruction mechanism for obtaining a boolean from the condition code. The compiler can emit the instruction

$$\texttt{i2i\_<}\quad \texttt{cc}_i\texttt{,r}_t\texttt{,r}_f \quad \Rightarrow \texttt{r}_l$$

Digression: Short-circuit evaluation as an optimization

Short-circuit evaluation arose naturally from a positional encoding of the value of boolean and relational expressions. On processors that used condition codes to record the result of a comparison and used conditional branches to interpret the condition code, short-circuiting made sense.

As processors include features like conditional move, boolean-valued comparisons, and predicated execution, the advantages of short-circuit evaluation will likely fade. With branch latencies growing, the cost of the conditional branches required for short-circuiting will grow. When the branch costs exceed the savings from avoiding evaluation, short circuiting will no longer be an improvement. Instead, full evaluation would be faster.

When the language requires short-circuit evaluation, as does C, the compiler may need to perform some analysis to determine when it is safe to substitute full evaluation for short-circuiting. Thus, future C compilers may include analysis and transformation to replace short-circuiting with full evaluation, just as compilers in the past have performed analysis and transformation to replace full evaluation with short circuiting.

where $\mathtt{r}_t$ is known to contain `true` and $\mathtt{r}_f$ is known to contain `false`. The effect of this instruction is to set $\mathtt{r}_1$ to `true` if condition code register $\mathtt{cc}_i$ has the value <, and to `false` otherwise.

The conditional move instruction executes in a single cycle. At compile time, it does not break a basic block; this can improve the quality of code produced by local optimization. At execution time, it does not disrupt the hardware mechanisms that prefetch and decode instructions; this avoids potential stalls due to mispredicted branches.

Boolean-valued Comparisons This scheme avoids the condition code entirely. The comparison operation returns a boolean value into either a general purpose register or into a dedicated, single-bit register. The conditional branch takes that result as an argument that determines its behavior.

The strength of this model lies in the uniform representation of boolean and relational values. The compiler never emits an instruction to convert the result of a comparison into a boolean value. It never executes a branch as part of evaluating a relational expression, with all the advantages ascribed earlier to the same aspect of conditional move.

The weakness of this model is that it requires explicit comparisons. Where the condition-code models can often avoid the comparison by arranging to have the condition code set by one of the arithmetic operations, this model requires the comparison instruction. This might make the code longer than under the condition branch model. However, the compiler does not need to have `true` and `false` in registers. (Getting them in registers might require one or two `loadI`s.)

Predicated Execution The architecture may combine boolean-valued comparisons with a mechanism for making some, or all, operations conditional. This

Straight Condition Codes

```
       comp     ra,rb    ⇒ cc1
       cbr_LT   cc1      → L1,L2
L1 :   add      rc,rd    ⇒ ra
       br                → Lout
L2 :   add      re,rf    ⇒ ra
Lout : nop
```

Conditional Move

```
comp     ra,rb          ⇒ cc1
add      rc,rd          ⇒ rt1
add      re,rf          ⇒ rt2
i2i_<    cc1,rt1,rt2    ⇒ ra
```

Boolean Compare

```
       cmp_LT   ra,rb    ⇒ r1
       cbr      r1       → L1,L2
L1 :   add      rc,rd    ⇒ ra
       br                → Lout
L2 :   add      re,rf    ⇒ ra
Lout : nop
```

Predicated Execution

```
          cmp_LT   ra,rb   ⇒ r1
 (r1):    add      rc,rd   ⇒ ra
(¬r1):    add      re,rf   ⇒ ra
```

Figure 8.4: Relational Expressions for Control-Flow

technique, called *predicated execution*, lets the compiler generate code that avoids using conditional branches to evaluate relational expressions. In ILOC, we write a predicated instruction by including a predicate expression before the instruction. To remind the reader of the predicate's purpose, we typeset it in parentheses and follow it with a question mark. For example,

```
(r17)?   add   ra,rb   ⇒ rc
```

indicates an add operation (r_a+r_b) that executes if and only if r_{17} contains the value `true`. (Some architects have proposed machines that always execute the operation, but only assign it to the target register if the predicate is `true`. As long as the "idle" instruction does not raise an exception, the differences between these two approaches are irrelevant to our discussion.) To expose the complexity of predicate expressions in the text, we will allow boolean expressions over registers in the predicate field. Actual hardware implementations will likely require a single register. Converting our examples to such a form requires the insertion of some additional boolean operations to evaluate the predicate expression into a single register.

8.4.3 Choosing a Representation

The compiler writer must decide when to use each of these representations. The decision depends on hardware support for relational comparisons, the costs of branching (particularly a mispredicted conditional branch), the desirability of short-circuit evaluation, and how the result is used by the surrounding code.

Consider the following code fragment, where the sole use for `(a < b)` is to alter control-flow in an `if–then–else` construct.

Straight Condition Codes

```
      comp    r_a, r_b  ⇒ cc_1
      cbr_LT  cc_1      → L_1,L_2
L_1:  comp    r_c, r_d  ⇒ cc_2
      cbr_LT  cc_2      → L_3,L_2
L_2:  loadI   false     ⇒ r_x
      br                → L_4
L_3:  loadI   true      ⇒ r_x
      br                → L_4
L_4:  nop
```

Conditional Move

```
comp    r_a,r_b          ⇒ cc_1
i2i_<   cc_1,r_t,r_f     ⇒ r_1
comp    r_c,r_d          ⇒ cc_2
i2i_<   cc_2,r_t,r_f     ⇒ r_2
and     r_1,r_2          ⇒ r_x
```

Boolean Compare

```
cmp_LT  r_a, r_b  ⇒ r_1
cmp_LT  r_c, r_d  ⇒ r_2
and     r_1, r_2  ⇒ r_x
```

Predicated Execution

```
cmp_LT  r_a, r_b  ⇒ r_1
cmp_LT  r_c, r_d  ⇒ r_2
and     r_1, r_2  ⇒ r_x
```

Figure 8.5: Relational Expressions for Assignment

```
if (a < b)
   then a ← c + d
   else a ← e + f
```

Figure 8.4 shows the code that might be generated under each hardware model.

The two examples on the left use conditional branches to implement the `if-then-else`. Each takes five instructions. The examples on the right avoid branches in favor of some form of conditional execution. The two examples on top use an implicit representation; the value of `a < b` exists only in $\texttt{cc}_1$, which is not a general purpose register. The bottom two examples create an explicit boolean representation for `a < b` in $\texttt{r}_1$. The left two examples use the value, implicit or explicit, to control a branch, while the right two examples use the value to control an assignment.

As a second example, consider the assignment $\texttt{x} \leftarrow \texttt{a < b} \wedge \texttt{c < d}$. It appears to be a natural for a numerical representation, because it uses ∧ and because the result is stored into a variable. (Assigning the result of a boolean or relational expression to a variable necessitates a numerical representation, at least as the final product of evaluation.) Figure 8.5 shows the code that might result under each of the four models.

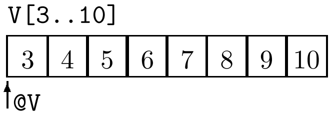
Again, the upper two examples use condition codes to record the result of a comparison, while the lower two use boolean values stored in a register. The left side shows the simpler version of the scheme, while the right side adds a form of conditional operation. The bottom two code fragments are shortest; they are identical because predication has no direct use in the chosen assignment. Conditional move produces shorter code than the straight condition code scheme. Presumably, the branches are slower than the comparisons, so the code is faster, too. Only the straight condition code scheme performs short-circuit evaluation.

8.5 Storing and Accessing Arrays

So far, we have assumed that variables stored in memory are scalar values. Many interesting programs use arrays or similar structures. The code required to locate and reference an element of an array is surprisingly complex. This section shows several schemes for laying out arrays in memory and describes the code that each scheme produces for an array reference.

8.5.1 Referencing a Vector Element

The simplest form of an array has a single dimension; we call a one-dimensional array a *vector*. Vectors are typically stored in contiguous memory, so that the i^{th} element immediately precedes the $i + 1^{st}$ element. Thus, a vector `V[3..10]` generates the following memory layout.



When the compiler encounters a reference, like `V[6]`, it must use the index into the vector, along with facts available from the declaration of `V`, to generate an offset for `V[6]`. The actual address is then computed as the sum of the offset and a pointer to the start of `V`, which we write as `@V`.

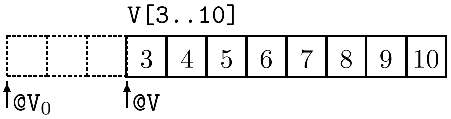
As an example, assume that `V` has been declared as `V[low..high]`, where `low` and `high` are the lower and upper bounds on the vector. To translate the reference `V[i]`, the compiler needs both a pointer to the start of storage for `V` and the offset of element `i` within `V`. The offset is simply $(\texttt{i} - \texttt{low}) \times w$, where w is the length of a single element of `V`. Thus, if `low` is 3 and `i` is 6, the offset is $(6 - 3) \times 4 = 12$. The following code fragment computes the correct address into $\texttt{r}_a$.

```
loadI  @i      ⇒ r1   // @i is i's address
subI   r1, 3   ⇒ r2   // (offset - lower bound)
multI  r2, 4   ⇒ r3   // × element length
addI   r3, @V  ⇒ r4   // @V is V's address
load   r4      ⇒ rv
```

Notice that the textually simple reference `V[i]` introduces three arithmetic operations. These can be simplified. Forcing a lower bound of zero eliminates the subtraction; by default, vectors in C have zero as their lower bound. If the element length is a power of two, the multiply can be replaced with an arithmetic shift; most element lengths have this property. Adding the address and offset seems unavoidable; perhaps this explains why most processors include an address mode that takes a *base address* and an *offset* and accesses the location at *base address* + *offset*.[2] We will write this as `loadAO` in our examples. Thus, there are obvious ways of improving the last two operations.

[2] Since the compiler cannot eliminate the addition, it has been folded into hardware.

If the lower bound for an array is known at compile-time, the compiler can fold the adjustment for the vector's lower bound into its address. Rather than letting @V point directly to the start of storage for V, the compiler can use $@V_0$, computed as $@V - \texttt{low} \times w$. In memory, this produces the following layout.



We sometimes call $@V_0$ the "false zero" of V. If the bounds are not known at compile-time, the compiler might calculate V_0 as part of its initialization activity and reuse that value in each reference to V. If each call to the procedure executes one or more references to V, this strategy is worth considering.

Using the false zero, the code for accessing V[i] simplifies to the following sequence:

```
loadI    @V_0      ⇒ r_{@V}   // adjusted address for V
load     @i        ⇒ r_1      // @i is i's address
lshiftI  r_1, 2    ⇒ r_2      // × element length
loadAO   r_{@V}, r_2 ⇒ r_V
```

This eliminates the subtraction by low. Since the element length, w, is a power of two, we also replaced the multiply with a shift. More context might produce additional improvements. If either V or i appears in the surrounding code, then $@V_0$ and i may already reside in registers. This would eliminate one or both of the loadi instructions, further shortening the instruction sequence.

8.5.2 Array Storage Layout

Accessing a multi-dimensional array requires more work. Before discussing the code sequences that the compiler must generate, we must consider how the compiler will map array indices into memory locations. Most implementations use one of three schemes: *row-major order*, *column-major order*, or *indirection vectors*. The source language definition usually specifies one of these mappings.

The code required to access an array element depends on the way that the array is mapped into memory. Consider the array A[1..2,1..4]. Conceptually, it looks like

A	1,1	1,2	1,3	1,4
	2,1	2,2	2,3	2,4

In linear algebra, the *row* of a two-dimensional matrix is its first dimension, and the *column* is its second dimension. In *row-major order*, the elements of A are mapped onto consecutive memory locations so that adjacent elements of a single row occupy consecutive memory locations. This produces the following layout.

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

The following loop nest shows the effect of row-major order on memory access patterns.

```
for $i \leftarrow 1$ to 2
   for $j \leftarrow 1$ to 4
      $A[i,j] \leftarrow A[i,j] + 1$
```

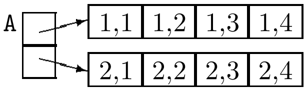
In row-major order, the assignment statement steps through memory in sequential order, beginning with `A[1,1]` and ending with `A[2,4]`. This kind of sequential access works well with most memory hierarchies. Moving the *i* loop inside the *j* loop produces an access sequence that jumps between rows, accessing `A[1,1]`, `A[2,1]`, `A[1,2]`, ..., `A[2,4]`. With a small array like `A`, this is not a problem. With larger arrays, the lack of sequential access could produce poor performance in the memory hierarchy. As a general rule, row-major order produces sequential access when the outermost subscript varies fastest.

The obvious alternative to row-major order is *column-major order.* It keeps the columns of `A` in contiguous locations, producing the following layout.

1,1	2,1	1,2	2,2	1,3	,2,3	1,4	2,4

Column major order produces sequential access when the innermost subscript varies fastest. In our doubly-nested loop, moving the *i* loop to the innermost position produces sequential access, while having the *j* loop inside the *i* loop produces non-sequential access.

A third alternative, not quite as obvious, has been used in several languages. This scheme uses *indirection vectors* to reduce all multi-dimensional arrays to a set of vectors. For our array `A`, this would produce



Each row has its own contiguous storage. Within a row, elements are addressed as in a vector (see Section 8.5.1). To allow systematic addressing of the row vectors, the compiler allocates a vector of pointers and initializes it appropriately.

This scheme appears simple, but it introduces two kinds of complexity. First, it requires more storage than the simpler row-major or column-major layouts. Each array element has a storage location; additionally, the inner dimensions require indirection vectors. The number of vectors can grow quadratically in the array's dimension. Figure 8.6 shows the layout for a more complex array, `B[1..2,1..3,1..4]`. Second, a fair amount of initialization code is required to set up all the pointers for the array's inner dimensions.

Each of these schemes has been used in a popular programming language. For languages that store arrays in contiguous storage, row-major order has been the typical choice; the one notable exception is Fortran, which used column-major order. Both BCPL and C use indirection vectors; C sidesteps the initialization issue by requiring the programmer to explicitly fill in all of the pointers.

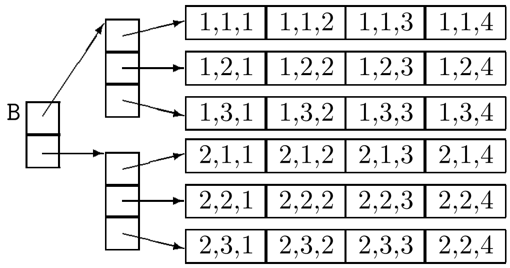


Figure 8.6: Indirection vectors for `B[1..2,1..3,1..4]`

8.5.3 Referencing an Array Element

Computing an address for a multi-dimensional array requires more work. It also requires a commitment to one of the three storage schemes described in Section 8.5.2.

Row-major Order In row-major order, the address calculation must find the start of the row and then generate an offset within the row as if it were a vector. Recall our example of `A[1..2,1..4]`. To access element `A[i,j]`, the compiler must emit code that computes the address of row `i`, and follow that with the offset for element j, which we know from Section 8.5.1 will be $(j - low_2) \times w$. Each row contains 4 elements, computed as $high_2 - low_2 + 1$, where $high_2$ is the the highest numbered column and low_2 is the lowest numbered column—the upper and lower bounds for the second dimension of `A`. To simplify the exposition, let $len_2 = high_2 - low_2 + 1$. Since rows are laid out consecutively, row `i` begins at $(i - low_1) \times len_2 \times w$ from the start of `A`. This suggests the address computation:

$$\texttt{@A} + (\texttt{i} - low_1) \times len_2 \times w + (\texttt{j} - low_2) \times w$$

Substituting actual values in for `i`, `j`, low_1, $high_2$, low_2, and w, we find that `A[2,3]` lies at offset

$$((2-1) \times 4 + (3-1)) \times 4 \;=\; 24$$

from `A[0,0]`. (`A` actually points to the first element, at offset 0.) Looking at `A` in memory, we find that `A[0,0]` + 24 is, in fact, `A[2,3]`.

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

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 adjust the base address in the two-dimensional case produces

$$\texttt{@A} + (\texttt{i} \times len_2 \times w) - (low_1 \times len_2 \times w) + (\texttt{j} \times w) - (low_2 \times w), \text{ or}$$

$$\texttt{@A} + (\texttt{i} \times len_2 \times w) + (\texttt{j} \times w) - (low_1 \times len_2 \times w + low_2 \times w)$$

The last term, $(low_1 \times len_2 \times w + low_2 \times w)$, is independent of `i` and `j`, so it can be factored directly into the base address to create

$$\texttt{@A}_0 = \texttt{@A} - (low_1 \times len_2 \times w + low_2 \times w)$$

This is the two-dimensional analog of the transformation that created a false zero for vectors in Section 8.5.1. Then, the array reference is simply

$$\texttt{@A}_0 + \texttt{i} \times len_2 \times w + \texttt{j} \times w$$

Finally, we can re-factor to move the w outside, saving extraneous multiplies.

$$\texttt{@A}_0 + (\texttt{i} \times len_2 + \texttt{j}) \times w$$

This form of the polynomial leads to the following code sequence:

```
load     @i         ⇒ r_i    // i’s value
load     @j         ⇒ r_j    // j’s value
loadI    @A_0       ⇒ r_@a   // adjusted base for A
multI    r_i, len_2 ⇒ r_1    // i × len_2
add      r_1, r_j   ⇒ r_2    // + j
lshiftI  r_2, 2     ⇒ r_3    // × 4
loadAO   r_@a, r_3  ⇒ r_v
```

In this form, we have reduced the computation to a pair of additions, one multiply, and one shift. Of course, some of `i`, `j`, and $\texttt{@A}_0$ may be in registers.

If we do not know the array bounds at compile-time, we must either compute the adjusted base address at run-time, or use the more complex polynomial that includes the subtractions that adjust for lower bounds.

$$\texttt{@A} + ((\texttt{i} - low_1) \times len_2 + \texttt{j} - low_2) \times w$$

In this form, the code to evaluate the addressing polynomial will require two additional subtractions.

To handle higher dimensional arrays, the compiler must generalize the address polynomial. In three dimensions, it becomes

$$\begin{gathered} base\_address \ + \ w \ \times \\ (((index_1 - low_1) \times len_2 + index_2 - low_2) \times len_3) + index_3 - low_3) \end{gathered}$$

Further generalization is straight forward.

Column-major Order Accessing an array stored in column-major order is similar to the case for row-major order. The difference in calculation arises from the difference in storage order. Where row-major order places entire rows in contiguous memory, column-major order places entire columns in contiguous memory. Thus, the address computation considers the individual dimensions in the opposite order. To access our example array, `A[1..2,1..4]`, when it is stored in column major order, the compiler must emit code that finds the starting address for column `j` and compute the vector-style offset within that column for element `i`. The start of column `j` occurs at offset $(\texttt{j} - low_2) \times len_1 \times w$ from the start of `A`. Within the column, element `i` occurs at $(\texttt{i} - low_1) \times w$. This leads to an address computation of

$$\texttt{@A} + ((\texttt{j} - low_2) \times len_1 + \texttt{i} - low_1) \times w$$

Substituting actual values for `i`, `j`, low_1, low_2, len_1, and w, `A[2,3]` becomes

$$\texttt{@A} + ((3-1)\times 2 + (2-1))\times 4 \;=\; 20,$$

so that `A[2,3]` is 20 bytes past the start of `A`. Looking at the memory layout from Section 8.5.2, we see that $\texttt{A} + 20$ is, indeed, `A[2,3]`.

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

The same manipulations of the addressing polynomial that applied for row-major order work with column-major order. We can also adjust the base address to compensate for non-zero lower bounds. This leads to a computation of

$$\texttt{@A}_0 + (\texttt{j}\times len_1 + i)\times w$$

for the reference `A[i,j]` when bounds are known at compile time and

$$\texttt{@A}_0 + ((j - low_2)\times len_1 + i - low_1)\times w$$

when the bounds are not known.

For a three-dimensional array, this generalizes to

$$\begin{gathered} base\_address + w\,\times \\ (((index_3 - low_3)\times len_2 + index_2 - low_2)\times len_1) + index_1 - low_1) \end{gathered}$$

The address polynomials for higher dimensions generalize along the same lines as for row-major order.

Indirection Vectors Using indirection vectors simplifies the code generated to access an individual element. Since the outermost dimension is stored as a set of vectors, the final step looks like the vector access described in Section 8.5.1. For `B[i,j,k]`, the final step computes an offset from `k`, the outermost dimension's lower bound, and the length of an element for `B`. The preliminary steps derive the starting address for this vector by following the appropriate pointers through the indirection vector structure.

Thus, to access element `B[i,j,k]` in the array `B` shown in Figure 8.6, the compiler would use `B`, `i`, and the length of a pointer (4), to find the vector for the subarray `B[i,*,*]`. Next, it would use that result, along with `j` and the length of a pointer to find the vector for the subarray `B[i,j,*]`. Finally, it uses the vector address computation for index `k`, and element length w to find `B[i,j,k]` in this vector.

If the current values for `i`, `j`, and `k` exist in registers $\texttt{r}_i$, $\texttt{r}_j$, and $\texttt{r}_k$, respectively, and that $\texttt{@B}_0$ is the zero-adjusted address of the first dimension, then `B[i,j,k]` can be referenced as follows.

```
loadI    @B_0       ⇒ r_@B   // assume zero-adjusted pointers
lshiftI  r_i, 2     ⇒ r_1    // pointer is 4 bytes
loadAO   r_@B, r_1  ⇒ r_2
lshiftI  r_j, 2     ⇒ r_3    // pointer is 4 bytes
loadAO   r_2, r_3   ⇒ r_4    // vector code from § 8.5.1
lshiftI  r_k, 2     ⇒ r_5
loadAO   r_4, r_5   ⇒ r_6
```

This code assumes that the pointers in the indirection structure have already been adjusted to account for non-zero lower bounds. If the pointers have not been adjusted, then the values in $\mathtt{r}_j$ and $\mathtt{r}_k$ must be decremented by the corresponding lower bounds.

Using indirection vectors, the reference requires just two instructions per dimension. This property made the indirection vector implementation of arrays efficient on systems where memory access was fast relative to arithmetic—for example, on most computer systems prior to 1985. Several compilers used indirection vectors to manage the cost of address arithmetic. As the cost of memory accesses has increased relative to arithmetic, this scheme has lost its advantage. If systems again appear where memory latencies are small relative to arithmetic, indirection vectors may again emerge as a practical way to decrease access costs.[3]

Accessing Array-valued Parameters When an array is passed as a parameter, most implementations pass it by reference. Even in languages that use call-by-value for all other parameters, arrays are usually passed by reference. Consider the mechanism required to pass an array by value. The calling procedure would need to copy each array element value into the activation record of the called procedure. For all but the smallest arrays, this is impractical. Passing the array as a reference parameter can greatly reduce the cost of each call.

If the compiler is to generate array references in the called procedure, it needs information about the dimensions of the array bound to the parameter. In Fortran, for example, the programmer is required to declare the variable using either constants or other formal parameters to specify its dimensions. Thus, Fortran places the burden for passing information derived from the array's original declaration to the called procedure. This lets each invocation of the procedure use the correct constants for the array that it is passed.

Other languages leave the task of collecting, organizing, and passing the necessary information to the compiler. This approach is necessary if the array's size cannot be statically determined—that is, it is allocated at run-time. Even when the size can be statically determined, this approach is useful because it abstracts away details that would otherwise clutter code. In these circumstances, the compiler builds a descriptor that contains both a pointer to the start of the array and the necessary information on each dimension. The descriptor has a known size, even when the array's size cannot be known at compile time. Thus, the compiler can allocate space for the descriptor in the AR of the called procedure. The value passed in the array's parameter slot is a pointer to this descriptor. For reasons lost in antiquity, we call this descriptor a *dope vector*.

When the compiler generates a reference to an array that has been passed as a parameter, it must draw the information out of the dope vector. It generates the same address polynomial that it would use for a reference to a local array, loading values out of the dope vector as needed. The compiler must decide, as a matter of policy, which form of the addressing polynomial it will use. With the naive address polynomial, the dope vector must contain a pointer to the start of

[3] On cache-based machines, locality is critical to performance. There is little reason to believe that indirection vectors have good locality. It seems more likely that this scheme generates a reference stream that appears random to the memory system.

```
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;
  begin;
    declare x float;
    declare i, j, k fixed binary;
    ... x = A(i,j,k);
    ...
  end fee;
```

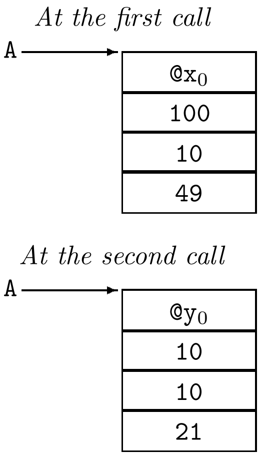


Figure 8.7: Dope Vectors

the array, the lower bound of each dimension, and all but one of the dimension sizes. With the address polynomial based on the false zero, the lower bound information is unnecessary. As long as the compiler always uses the same form of the polynomial, it can generate code to build the dope vectors as part of the prologue code for the procedure call.

A given procedure can be invoked from multiple call sites. At each call site, a different array might be passed. The PL/I fragment in figure 8.7 illustrates this. The program `main` contains two statements that call `fee`. The first passes array `x`, while the second passes `y`. Inside `fee`, the actual parameter (`x` or `y`) is bound to the formal parameter `A`. To allow the code for `fee` to reference the appropriate location, it needs a dope vector for `A`. The respective dope vectors are shown on the right hand side of the figure.

As a subtle point, notice that the cost of accessing an array-valued parameter is higher than the cost of accessing an array declared locally. At best, the dope vector introduces additional memory references to access the relevant entries. At worst, it prevents the compiler from performing certain optimizations that rely on complete knowledge of the array's declaration.

8.5.4 Range Checking

Most programming language definitions assume, either explicitly or implicitly, that a program only refers to array elements within the defined bounds of the array. A program that references an out-of-bounds element is, by definition, not well formed. Many compiler-writers have taken the position that the compiler should detect out-of-bounds array accesses and report them in a graceful fashion to the user.

The simplest implementation of range checking inserts a test before each array reference. The test verifies that each index value falls in the valid range for the dimension in which it will be used. In an array-intensive program, the overhead of such checking can be significant. Many improvements on this simple scheme are possible.

If the compiler intends to perform range checking on array-valued parameters, it may need to include additional information in the dope vectors. For example, if the compiler uses the address polynomial based on the array's false zero, it will have lengths for each dimension, but not upper and lower bound information. An imprecise test might be done by checking the offset against the total array size; to perform the precise test would require passing upper and lower bounds for each dimension.

When the compiler generates run-time code for range checking, it must insert many copies of the code that reports the error. Typically, this involves a branch to a run-time error routine. During normal execution, these branches are rarely taken; if the error handler stops execution, then it can run at most once execution. If the target machine provides a mechanism that lets the compiler predict the likely direction of a branch, these exception branches should be predicted as not taken. Furthermore, the compiler may want to annotate its internal representation to show that these branches lead to an abnormal termination. This allows subsequent phases of the compiler to differentiate between the "normal" execution path and the "error" path; the compiler may be able to use this knowledge to produce better code along the non-error path.

8.6 Character Strings

The operations provided for character-based data are often quite different from those provided for string data. The level of programming language support ranges from C, where most manipulation takes the form of calls to library routines, to PL/I, where assignment of individual characters, arbitrary substrings of characters, and even concatenation of strings occur as first-class operators in the language. To present the issues that arise in string implementation, this section discusses the implementation of assigning substrings, of concatenating two strings, and of computing a string's length.

String operations can be costly. Older CISC architectures, such as the IBM S/370 and the DIGITAL VAX, provided strong support for string manipulation. Modern RISC machines rely more heavily on the compiler to encode these complex operations into a set of simpler interactions. The basic operation, copying bytes from one location to another, arises in many different contexts.

8.6.1 String Representation

The compiler writer must choose a representation for strings; the details of the string representation have a strong impact on the cost of various string operations.

Consider, for example, the difference between the two possible string representations. The one on the left is used by C. It uses a simple vector of characters, with a designated character as a terminator. The representation on the right stores the length of the string (8) alongside its contents.

a	b̸	s	t	r	i	n	g	⊥	

Null-termination

8	a	b̸	s	t	r	i	n	g

Explicit length field

Storing the length increases the size of the string in memory. However, it simplifies several operations on strings. For fixed length strings, both Scheme and PL/I use the length format. When the language allows varying length strings to be stored inside a string allocated to some fixed length, the implementor might also store the allocated length with the string. This allows the compiler to implement run-time checking for overrunning the string length on assignment and concatenation.

8.6.2 String Assignment

String assignment is conceptually simple. In C, an assignment from the third character of b to the second character of a can be written as shown on the left:

```
                  loadI     @b     ⇒ r_b
                  cloadAI   r_b,2  ⇒ r_2
a[1] = b[2];      loadI     @a     ⇒ r_a
                  cstoreAI  r_2    ⇒ r_a,1
```

With appropriate support, this would translate directly into the code shown on the right. It uses the operations from the cload and cstore family to perform single character accesses. (Recall that a[0] is the first character in a, because C uses a default lower bound of zero.)

If, however, the underlying hardware does not support character-oriented memory operations, the compiler must generate more complex code. Assuming that both a and b begin on word boundaries, the compiler might emit the following code:

```
loadI    @b          ⇒ r_b     // get word
load     r_b         ⇒ r_1
loadI    0x0000FF00  ⇒ r_2     // mask for 3rd char
and      r_1,r_2     ⇒ r_3     // ‘and’ away others
loadI    8           ⇒ r_4
lshift   r_3,r_4     ⇒ r_5
loadI    @a          ⇒ r_a     // want 1st word
load     r_a         ⇒ r_6
loadI    0xFF00FFFF  ⇒ r_7     // mask for 2nd char
and      r_6,r_7     ⇒ r_8
and      r_3,r_8     ⇒ r_9     // new 1st word of a
storeAI  r_9         ⇒ r_a,0   // put it back
```

This loads the appropriate word from b, extracts the desired character, shifts it to a new position, masks it into the appropriate word from a, and stores the result back into a.

With longer strings, the code is similar. PL/I has a string assignment operator. The programmer can write an operation such as `a = b;`, where `a` and `b` have been declared as character strings. Assuming that `a` has enough room to hold `b`, the following simple loop will move the characters on a machine with byte-oriented `load` and `store` operations:

```
      loadI     1       ⇒ r1        // set up for loop
      loadI     0       ⇒ r2
      i2i       r2      ⇒ r3
      loadI     @b      ⇒ rb
      loadAI    rb,-4   ⇒ r5        // get b's length
      loadI     @a      ⇒ ra
      loadAI    ra,-4   ⇒ r6        // get a's length
      cmp_LT    r6,r5   ⇒ r7
      cbr       r7      → Lsov,L1   // raise error ?
L1:   cmp_LE    r2,r5   ⇒ r8        // more to copy ?
      cbr       r8      → L2,L3
L2:   cloadAO   rb,r2   ⇒ r9        // get char from b
      cstoreAO  r9      ⇒ ra,r2     // put it in a
      add       r2,r1   ⇒ r2        // increment offset
      cmp_LE    r2,r5   ⇒ r10       // more to copy ?
      cbr       r10     → L2,L3
L3:   nop                           // next statement
```

Notice that this code tests the length of `a` and `b` to avoid overrunning `a`. The label L_{sov} represents a run-time error handler for string overflow conditions.

With null-terminated strings, the code changes somewhat:

```
    loadI   @b     ⇒ rb      // get pointers
    loadI   @a     ⇒ ra
    loadI   1      ⇒ r1      // the increment
    loadI   NULL   ⇒ r2      // EOS char
    cload   rb     ⇒ r3      // get 1st char
    cmp_NE  r2,r3  ⇒ r4      // test it
    cbr     r4     → L1,L2
L1: cstore  r3     ⇒ ra      // store it
    add     rb,r1  ⇒ rb      // bump pointers
    add     ra,r1  ⇒ ra
    cload   rb     ⇒ r3      // get next char
    cmp_NE  r2,r3  ⇒ r4
    cbr     r4     → L1,L2
L2: nop                      // next statement
```

This code implements the classic character copying loop used in C programs. It does not test for overrunning `a`. That would require a computation of the length of both `a` and `b` (see Section 8.6.4).

```
    loadI     @b          ⇒ rb       // set up the loop
    loadAI    rb,-4       ⇒ r2       // get b's length
    loadI     0xFFFFFFFC  ⇒ r3       // mask for full word
    and       r2,r3       ⇒ r4       // length(b), masked
    loadI     4           ⇒ r5       // increment
    loadI     0           ⇒ r6       // offset in string
    loadI     @a          ⇒ ra
    loadAI    ra,-4       ⇒ r20      // get a's length
    cmp_LT    r20,r2      ⇒ r21
    cbr       r21         → Lsov,L1
L1: cmp_LE    r5,r2       ⇒ r7       // more for the loop?
    cbr       r7          → L2,L3
L2: loadAO    rb,r6       ⇒ r9       // get char from b
    storeAO   r9          ⇒ ra,r6    // put it in a
    add       r6,r5       ⇒ r6       // increment offset
    cmp_LT    r6,r4       ⇒ r10      // more for the loop?
    cbr       r10         → L2,L3
L3: sub       r2,r4       ⇒ r11      // # bytes to move
    loadAO    rb,r6       ⇒ r9       // get last byte
    loadI     @MASK1      ⇒ r12      // get Source mask
    and       r9,r12      ⇒ r13
    loadI     @MASK2      ⇒ r14      // get Destination mask
    loadAO    ra,r6       ⇒ r15      // get Destination word
    and       r15,r14     ⇒ r16      // mask out destination
    and       r16,r13     ⇒ r17      // combine them
    storeAO   r17         ⇒ ra,r6    // put it in a
```

Figure 8.8: String assignment using whole-word operations

```
while (*b != '\0')
  *a++ = *b++;
```

With hardware support for autoincrement on `load` and `store` operations, the two `add`s in the loop would occur during the `cload` and `cstore` operations. This reduces the loop to four operations. (Recall that C was designed for the PDP/11, which supported auto-post-increment.) Without autoincrement, the compiler would generate better code by using `cloadAO` and `cstoreAO` with a common offset. That would require only one `add` operation inside the loop.

Without byte-oriented memory operations, the code becomes more complex. The compiler could replace the `load`, `store`, `add` portion of the loop body with the scheme for masking and shifting single characters into the body of the loop shown earlier. The result is a functional, but ugly, loop. This increases substantially the number of instructions required to move `b` into `a`.

The alternative is to adopt a somewhat more complex scheme that makes whole-word `load` and `store` operations an advantage rather than a burden. The compiler can use a word-oriented loop, followed by a post-loop clean-up operation that handles any leftover characters at the end of the string. Figure 8.8 shows one way of implementing this.

The code required to set up the loop is more complex, because it must compute the length of the substring that can be moved with whole-word operations ($\lceil \mathit{length}(\texttt{b})/4 \rceil$). Once again, the loop body contains five instructions. However, it uses just one quarter of the iterations used by the character-oriented loop. The post-loop code that handles any remaining bytes relies on the presence of two global arrays:

MASK1	
Index	*Value*
0	0x00000000
1	0xFF000000
2	0xFFFF0000
3	0xFFFFFF00

Source Mask

MASK2	
Index	*Value*
0	0xFFFFFFFF
1	0x00FFFFFF
2	0x0000FFFF
3	0x000000FF

Destination Mask

Using these arrays allows the code to handle one, two, or three leftover bytes without introducing further branches.

Of course, if the compiler knows the length of the strings, it can avoid generating the loop and, instead, emit the appropriate number of `loads`, `stores`, and `adds`.

So far, we have assumed that both the source and destination strings begin on word-aligned boundaries. The compiler can ensure that each string begins on a word-aligned boundary. However, arbitrary assignments create the need for code to handle arbitrary alignment of both the source and the destination strings. With character-oriented memory operations, the same code works for arbitrary character-oriented alignment. With word-oriented memory operations, the compiler needs to emit a pre-loop code that brings both the source and the destination to word-aligned boundaries, followed by a loop that `loads` a word from the source, shuffles the characters into place for the destination, and `stores` a word into the destination. Of course, the loop is followed by code to clean up the final one, two, or three bytes.

8.6.3 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`.

The former case is a length computation followed by an assignment. The compiler emits code to determine the length of `a`, and performs an assignment of `b` to the space that immediately follows the contents of `a`.

The latter case requires copying each character in `a` and each character in `b`.

The compiler treats the concatenation as a pair of assignments and generates code as shown in Section 8.6.2.

In either case, the compiler should ensure that `length(a||b)` is not greater the space allocated to hold the result. In practice, this requires that either the compiler or the run-time system record the allocated length of each string. If the lengths are known at compile-time, the compiler can perform the check during code generation and avoid emitting code for a run-time check. Often, however, the compiler cannot know the length of `a` and `b`, so it must generate code to compute the lengths at run-time and to perform the appropriate test and branch.

8.6.4 String Length

Some applications need to compute the length of a character string. In C programs, the function `strlen` in the standard library takes a string as its argument and returns the string's length, expressed as an integer. In PL/I, the built-in function `length` performs the same function. The two string representations described earlier lead to radically different costs for the length computation.

Null-terminated string The length computation must start at the beginning of the string and examine each character, in order, until it reaches the null character. The code is quite similar to the C character copying loop. This requires time proportional to the length of the string.

Explicit length field The length computation is a memory reference. In ILOC, this requires a `loadI` of the string address and a `loadAI` to obtain the length. The cost is constant and small.

The tradeoff between these representations is simple; null-termination saves space, while an explicit length field makes the length computation inexpensive.

The classic example of a string optimization problem is reporting the length that would result from the concatenation of two strings, *a* and *b*. In a PL/I-like notation, this would be written as `length(`*a*`||`*b*`)`. In C, it would be written as either `strlen(strcat(`*a*`,`*b*`))` or `strlen(`*a*`)+strlen(`*b*`)`. The desired solution avoids building the concatenated string to measure its length. The two distinct ways of writing it in C expose the difference and leave it up to the programmer to determine which is used. Of course, with C's representation for the string, the computation must still touch each character in each string. With a PL/I-style representation, the operation can be optimized to use two `loads` and an `add`. (Of course, the programmer could directly write the form that produced efficient code—`length(`*a*`)+length(`*b*`)` in PL/I.)

8.7 Structure References

The other kind of complex data structure that occurs in most programming languages is a structure, or some variation on it. In C, a structure aggregates together individually named elements, often of differing types. A list implementation, in C, might use the structures

```
union Node
{
  struct ValueNode;
  struct ConstructorNode;
};

struct ValueNode
{
  int Value;
};
```

```
struct ConstructorNode
{
  Node *Head;
  Node *Tail;
};

ValueNode NilNode;
Node* NIL = &NilNode;
```

Using these two nodes, the programmer can construct LISP-like S-expressions. A `ConstructorNode` can point to a pair of `Nodes`. A `Node` can be either a `ValueNode` or a `ConstructorNode`.

Working with structures in C requires the use of *pointer values*. In the declaration of `ConstructorNode`, both `Head` and `Tail` are pointers to data items of type `Node`. The use of pointers introduces two distinct problems for the compiler: *anonymous objects* and *structure layout.*

Loading and Storing Anonymous Values A C program creates an instance of a structure in one of two ways. It can declare a structure instance; `NilNode` in the example above is a declared instance of `ValueNode`. Alternatively, it can dynamically allocate a structure instance. In `c`, this looks like:

```
NIL = (NODE *) malloc(sizeof(ValueNode));
```

The instance of `ValueNode` that this creates has no variable name associated with it—the only access is through a pointer.

Because the only "name" for an anonymous value is a pointer, the compiler cannot easily determine whether or not two pointer references specify the same memory location. Consider the code fragment.

```
1.  p1 = (NODE *) malloc(sizeof(ConstructorNode));
2.  p2 = (NODE *) malloc(sizeof(ConstructorNode));
3.  if (...)
4.     then p3 = p1;
5.     else p3 = p2;
6.  p1->Head = ...;
7.  p3->Head = ...;
8.    ...    = p1->Head;
```

The first two lines create anonymous `ConstructorNodes`. Line six initializes the node reachable through `p1`. Line seven either initializes the node reachable through `p2` or overwrites the value recorded by line six. Finally, line eight references the value stored in `p1->Head`.

To implement the sequence of assignments in lines six through eight, the compiler would like to keep the value that is reused in a register. Unfortunately, the compiler cannot easily determine whether line eight refers to the value generated in line six or the value generated in line seven. To understand the answer to that question, the compiler must be able to determine, with certainty, the value of the conditional expression evaluated in line three. While this may be possible in certain specific instances (*i.e.*, `1>2`), it is undecidable in the general case. Thus, the compiler must emit conservative code for this sequence of assignments. It must `load` the value used in line eight from memory, even though it had the value in a register quite recently (in either line six or line seven).

This degree of uncertainty that surrounds references to anonymous objects forces the compiler to keep values used in pointer-based references in memory. This can make statements involving pointer-based references less efficient than corresponding computations on local variables that are stored unambiguously in the procedure's AR. Analyzing pointer values and using the results of that analysis to disambiguate references—that is, to rewrite references in ways that let the compiler keep some values in registers—is a major source of improvement for pointer intensive programs. (A similar effect occurs with code that makes intensive use of arrays. Unless the compiler performs an in-depth analysis of the array subscripts, it may not be able to determine whether or not two array references overlap. When the compiler cannot distinguish between two references, such as `a[i,j,k]` and `a[i,j,l]`, it must treat both references conservatively.)

Understanding Structure Layouts To emit code for structure references, the compiler must know both the starting address of the structure instance and the offset and length of each element. To maintain this information, the compiler typically builds a separate table of structure layouts. The table must include the textual name for each structure element, its offset within the structure, and its source-language data type. For the list example, the compiler might build the following structures:

Structure Table		
Name	*Length*	*1^{st} El't*
`ValueNode`	4	0
`ConstructorNode`	8	1

Element Table				
Name	*Length*	*Offset*	*Type*	*Next*
`ValueNode.Value`	4	0	`int`	⊥
`ConstructorNode.Head`	4	0	`Node *`	2
`ConstructorNode.Tail`	4	4	`Node *`	⊥

Entries in the element table use fully qualified name. This avoids conflicts due to reuse of a name in several distinct structures. (Few languages insist that programs use unique element names inside structures.)

With this information, the compiler can easily generate code for structure references. The reference `p1->Head` might translate into the ILOC sequence

```
loadI    0        ⇒ r1   // offset of 'Head'
loadAO   rp1,r1   ⇒ r2   // value of p1->Head
```

Here, the compiler found the offset of `Head` by following the table from the `ConstructorNode` entry in the structure table to the `Head` entry in the element table. The start address of `p1->Head` resides in `p1`.

Many programming languages allow the user to declare an array of structures. If the user is allowed to take the address of a structure-valued element of this array, then the compiler must lay out the data in memory as multiple copies of the structure layout. If the programmer cannot take the address of a structure-valued element of the array, the compiler might lay out the structure as if it were a structure composed of elements that were, themselves, arrays. Depending on how the surrounding code accesses the data, these two strategies might have strikingly different performance on a system with cache memory.

To address an array of structures, laid out as multiple copies of the structure, the compiler uses the array address polynomials described in the previous section. The overall length of the structure becomes the element size, w, in the address polynomial. The polynomial generates the address of the start of the structure instance. To obtain the value of a specific element, the element's offset is added to the instance address.

If the compiler has laid out the structure with elements that are arrays, it must compute the starting location of the element array using the offset table information and the array dimension. This address can then be used as the starting point for an address calculation using the appropriate polynomial.

Unions and Run-time Tags For notational convenience, some programming languages allow union types. This allows a single memory location to be interpreted in different ways. The `Node` declaration given earlier allows a single pointer to refer to an object of type `ValueNode` or an object of type `ConstructorNode`. The meaning of any reference is clear, because the two structures have no common element names.

In general, the compiler (and the programming language) must make provision for the case when the meaning of a union-type reference is unclear. In practice, two alternatives arise: additional syntax and run-time tags. The language can place the burden for disambiguating union references on the programmer, by requiring fully qualified names. In the `Node` example, the programmer would need to write `p1->ConstructorNode.Head` or `p2->ValueNode.Value`.

The alternative is to include a run-time tag in each allocated instance of the union. PL/I required that the programmer explicitly declare the tag and its values. Other systems have relied on the translator to insert both the run-time tag and the appropriate code to check for correct use at each reference to an anonymous object. In fact, much of the practical motivation for stronger type systems and compile-time algorithms that can prove type-correctness arose from the desire to eliminate these automatically generated checks on run-time tags. The overhead of run-time tag checking can be significant.

8.8 Control Flow Constructs

A maximal-length sequence of assignment statements forms a basic block. Almost any other executable statement causes a control-flow transfer that ends the preceding block, as does a statement that can be the target of a branch. As the compiler generates code, it can build up basic blocks by simply aggregating together consecutive assignment statements. If the generated code is held in a simple linear array, each block can be described by a tuple, ⟨*first,last*⟩, that holds the indices of the instruction that begins the block and the instruction that ends the block.

To construct an executable program from the set of blocks, the compiler needs to tie the blocks together with code that implements the control-flow operations of the source program. To capture the relationship between the blocks, many compilers build a control-flow graph (CFG, see Section 6.3.4) that gets used for analysis, optimization, and code generation. In the CFG, nodes represent basic blocks and edges represent possible transfers of control between blocks. Typically, the CFG is a lightweight representation that contains references to a more detailed representation of each block.

Beyond basic blocks, the compiler must generate code for the control-flow constructs used in the source language. While many different syntactic conventions have been used to express control-flow, the number of underlying constructs is small. This section shows the kind of code that the compiler should generate for most of the control-flow constructs found in modern programming languages.

8.8.1 Conditional Execution

Most programming languages provide the functionality of the `if-then-else` construct. Given the source text

```
if expr
    then statement₁
    else statement₂
```

the compiler must generate code that evaluates *expr* and branches to either *statement*$_1$ or *statement*$_2$ based on the value of *expr*. As we saw in Section 8.4, the compiler has many options for implementing `if-then-else` constructs.

Most languages evaluate the controlling expression, *expr*, to a boolean value. If it has the value `true`, the statement under the `then` part executes. Otherwise, the statement under the `else` part executes. The earlier discussion focussed on evaluating the controlling expression; it showed how the underlying instruction set influenced the strategies for handling both the controlling expression and, in some cases, the controlled statements. It assumed that the code under the `then` and `else` parts was reasonably compact; most of the examples translated into a single instruction.

In practice, programmers can place arbitrarily large code fragments inside the `then` and `else` parts. The size of these code fragments has an impact on the compiler's strategy for implementing the `if–then–else` construct. With trivial

Using Predicates

Unit 1	*Unit 2*
comparison ⇒ $\texttt{r}_1$	
$(\texttt{r}_1)$ $\texttt{op}_1$	$(\neg\texttt{r}_1)$ $\texttt{op}_2$
$(\texttt{r}_1)$ $\texttt{op}_3$	$(\neg\texttt{r}_1)$ $\texttt{op}_4$
$(\texttt{r}_1)$ $\texttt{op}_5$	$(\neg\texttt{r}_1)$ $\texttt{op}_6$
$(\texttt{r}_1)$ $\texttt{op}_7$	$(\neg\texttt{r}_1)$ $\texttt{op}_8$
$(\texttt{r}_1)$ $\texttt{op}_9$	$(\neg\texttt{r}_1)$ $\texttt{op}_{10}$
$(\texttt{r}_1)$ $\texttt{op}_{11}$	$(\neg\texttt{r}_1)$ $\texttt{op}_{12}$
$(\texttt{r}_1)$ $\texttt{op}_{13}$	$(\neg\texttt{r}_1)$ $\texttt{op}_{14}$
$(\texttt{r}_1)$ $\texttt{op}_{15}$	$(\neg\texttt{r}_1)$ $\texttt{op}_{16}$
$(\texttt{r}_1)$ $\texttt{op}_{17}$	$(\neg\texttt{r}_1)$ $\texttt{op}_{18}$
$(\texttt{r}_1)$ $\texttt{op}_{19}$	$(\neg\texttt{r}_1)$ $\texttt{op}_{20}$

Using Branches

	Unit 1	*Unit 2*
	compare & branch	
$\texttt{L}_1$:	$\texttt{op}_1$	$\texttt{op}_3$
	$\texttt{op}_5$	$\texttt{op}_7$
	$\texttt{op}_9$	$\texttt{op}_{11}$
	$\texttt{op}_{13}$	$\texttt{op}_{15}$
	$\texttt{op}_{17}$	$\texttt{op}_{19}$
	br → $\texttt{L}_{out}$	
$\texttt{L}_2$:	$\texttt{op}_2$	$\texttt{op}_4$
	$\texttt{op}_6$	$\texttt{op}_8$
	$\texttt{op}_{10}$	$\texttt{op}_{12}$
	$\texttt{op}_{14}$	$\texttt{op}_{16}$
	$\texttt{op}_{18}$	$\texttt{op}_{20}$
	br → $\texttt{L}_{out}$	
$\texttt{L}_{out}$:	nop	

Figure 8.9: Predication versus Branching

then and else parts, as shown in Figure 8.4, the primary consideration for the compiler is matching the expression evaluation to the underlying hardware. As the then and else parts grow, the importance of efficient execution inside the then and else parts begins to outweigh the cost of executing the the controlling expression.

For example, on a machine that supports predicated execution, using predicates for large blocks in the then and else parts can waste execution cycles. Since the processor must issue each predicated instruction to one of its functional units, the cost of a non-executed instruction is roughly the same as that of an executed instruction. For an if–then–else construct with large blocks of code under both the then and else parts, the cost of unexecuted instructions may outweigh the overhead of using a conditional branch.

Figure 8.9 illustrates this tradeoff. The figure assumes that both the then and else parts contain ten independent ILOC operations, and that the target machine can issue two instructions per cycle.

The left side shows code that might be generated using predication; it assumes that the code evaluated the controlling expression into $\texttt{r}_1$. The code issues two instructions per cycle. One of those executes in each cycle. The code avoids all branching. If each operation takes a single cycle, it takes ten cycles to execute the controlled statements, independent of which branch is taken.

The right side shows code that might be generated using branches; it assumes that control flows to $\texttt{L}_1$ for the then part and to $\texttt{L}_2$ for the else part. Because the instructions are independent, the code issues two instructions per cycle. Following the then path, it takes five cycles to execute the operations for the taken path, plus the cost of the terminating branch. The cost for the else part

Digression: Branch Prediction by Users

One story that has achieved the status of an urban legend concerns branch prediction. Fortran has an arithmetic *if* statement that takes one of three branches, based on whether the controlling expression evaluates to a negative number, to zero, or to a positive number. One early IBM compiler allowed the user to supply weights for each label that reflected the relatively probability of taking that branch. The compiler then used the weights to order the branches in a way that minimized total expected delay from branching.

After the compiler had been in the field for some time, the story goes, a maintainer discovered that the branch weights were being used in the reverse order—maximizing the expected delay. No one had complained. The story is usually told as a moral fable about the value of a programmers' opinions about 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.)

is identical.

The predicated version avoids the initial conditional branch, as well as the terminating branch. The branching version incurs the overhead of both branches, but may execute faster. Each path contains a conditional branch, five cycles of operations, and the terminal branch (which is easily predicted and should have minimal cost.) The difference lies in the effective issue rate—the branching version issues roughly half the instructions of the predicated version. As the code fragments in the `then` and `else` parts grow larger, this difference becomes larger.

Choosing between branching and predication to implement an `if–then–else` requires some care. Several issues should be considered.

1. *expected frequency of execution for each part:* If one side of the conditional is expected to execute significantly more often, techniques that speed execution of that path may produce faster code. This bias may take the form of predicting a branch, of executing some instructions speculatively, or of reordering the logic.

2. *uneven amounts of code:* If one path through the construct is contains many more instructions than the other, this may weigh against predication (unless it is infrequently executed).

3. *control-flow inside the construct:* If either path through the construct contains non-trivial control flow, such as another `if–then–else`, a loop, a case statement, or procedure call, predication may not be the most efficient choice. In particular, nested `if` constructs create more complex predicate expressions and lower the percentage of issued instructions that are actually executed.

To make the best decision, the compiler must consider all these factors, as well as the surrounding context. These factors may be difficult to assess early in compilation; for example, optimization may change them in significant ways.

8.8.2 Loops and Iteration

Most programming languages include a control-flow construct to perform iteration. These loops range from the original Fortran `do` loop through C's `for` loop, `while` loop, and `until` loop. All of these loops have a common structure; a well-defined condition controls the iteration of the loop's body. The basic layout of these loops is as follows:

1. evaluate the controlling expression
2. if `false`, branch beyond the end of the loop
   otherwise, fall into the loop's body
3. at the end of the loop body, re-evaluate the controlling expression
4. if `true`, branch to the top of the loop body
   otherwise, fall through to the next statement

If the loop body contains no other control-flow, this produces a loop body with only one branch. The latency of this branch might be hidden in one of two ways. If the architecture allows the compiler to predict whether or not the branch is taken, the compiler should predict the loop-ending branch as being taken back to the next iteration. If the architecture allows the compiler to move instructions into the delay-slot(s) of the branch, the compiler should attempt to fill the delay slot(s) with instructions from the loop body.

For Loops and Do Loops To create a `for` loop, the compiler follows the layout given previously. This produces a simple loop with two distinct sections: a pre-loop section and a loop body. The pre-loop section initializes the induction variable and performs the initial evaluation and test of the controlling expression. The loop body implements the statements inside the loop, followed by the increment step from the loop's header and an evaluation and test of the controlling expression. Thus, the C code on the left might result in the ILOC code on the right.

```
for (i=1; i<=100; i++)
{
  loop body
}
```

```
        loadI    1        ⇒ r1
        loadI    1        ⇒ r2
        loadI    100      ⇒ r3
        cmp_GT   r1,r3    ⇒ r4
        cbr      r4       → L2,L1
L1:     loop body
        add      r1,r2    ⇒ r1
        cmp_LE   r1,r3    ⇒ r6
        cbr      r6       → L1,L2
L2:     next statement
```

If the compiler applies further transformations to the loop body, such as value numbering or instruction scheduling, the fact that the body is a single basic block may lead to better optimization.

The alternative is to use an absolute branch at the bottom of the loop that targets the update and conditional branch at the top of the loop body. This avoids replicating the update and conditional branch. However, it creates a two-block loop for even the simplest loops, and it typically lengthens the path through the loop by at least one operation. However, if code size is a serious consideration, then consistent use of this more compact loop form might be worthwhile. The loop-ending branch is unconditional and, thus, trivially predicted. Many modern processors avoid latency on unconditional branches by prefetching their targets.

The code for a Fortran `do` loop has a similar form, except for one odd quirk. The Fortran standard specifies that the number of iterations that a `do` loop makes is completely determined before the loop begins execution. Modifications to the induction variable made inside the loop body have no effect on the number of times the loop iterates.

```
   do 10 i = 1, 100
      ...
      i = i * 2
10    continue
```

```
      loadI   1       ⇒ r1
      loadI   1       ⇒ r2
      loadI   100     ⇒ r3
      loadI   1       ⇒ r4
      loadI   2       ⇒ r5
      cmp_GT  r4,r3   ⇒ r6
      cbr     r6      → L2,L1
L1:   loop body
      mult    r1,r5   ⇒ r1
      add     r1,r2   ⇒ r1
      add     r4,r2   ⇒ r4
      cmp_LE  r4,r3   ⇒ r7
      cbr     r7      → L1,L2
L2:   next statement
```

According to the Fortran standard, this example loop should execute its body one hundred times, despite the modifications to `i` inside the loop. To ensure this behavior, the compiler may need to generate a hidden induction variable, r_4, to control the iteration. This extra variable is sometimes called a *shadow variable.*

Unless the compiler can determine that the loop body does not modify the induction variable, the compiler must generate a shadow variable. If the loop contains a call to another procedure and passes the induction variable as a call-by-reference parameter, the compiler must assume that the called procedure modifies the induction variable, unless the compiler can prove otherwise.[4]

[4] This is one case where analyzing the entire program (for example, with interprocedural data-flow analysis) routinely wins. Programmers pass induction variables as parameters so that they can include the induction value in debugging output. Interprocedural analysis easily recognizes that this does not change the induction variable's value, so the shadow variable is unnecessary.

While Loops A `while` loop follows the same basic form, without the introduced overhead of an induction variable. The compiler emits code to evaluate the condition before the loop, followed by a branch that bypasses the loop's body. At the end of the loop, the condition is re-evaluated and a conditional branch takes control back to the top of the loop.

```
                              cmp_GE  rx,ry   ⇒ r1
    while (x < y)             cbr     r1      → L2,L1
    {
      loop body          L1:  loop body
    }                         cmp_LT  rx,ry   ⇒ r2
                              cbr     r2      → L1,L2
                         L2:  next statement
```

Again, replicating the evaluation and test at the end of the loop creates a single basic block for the body of a simple loop. The same benefits that accrue to a `for` loop from this structure occur with a `while` loop.

Until Loops For an `until` loop, the compiler generates code similar to the `while` loop. However, it omits the first evaluation and test. This ensures that control-flow always enters the loop; the test at the bottom of the loop body handles the `until` part of the test.

```
    until (x < y)        L1:  loop body
    {                         cmp_LT  rx,ry   ⇒ r2
      loop body               cbr     r2      → L1,L2
    }                    L2:  next statement
```

The `until` loop is particularly compact, because it has no pre-loop sequence.

Expressing Iteration as Tail Recursion In Lisp-like languages, iteration is often implemented (by programmers) using a stylized form of recursion. If the last action of a function is a call, that call is considered a *tail call.* If the tail call is a self-recursion, the call is considered a *tail recursion.* For example, to find the last element of a list in Scheme, the programmer might write the following simple function:

```
(define (last alon)
   (cond
      ((empty?  alon) empty)
      ((empty?  (rest alon)) (first alon))
      (else (last (rest alon)))))
```

Its final act is a tail-recursive call; this particular form of call can be optimized into a branch back to the top of the procedure. This avoids the overhead of a fully general procedure call (see Section 8.9). The effect is to replace a procedure call with a branch that binds some parameters. It avoids most of the operations of the procedure call and completely eliminates the space penalty for creating new activation records on each tail call. The results can rival a `for` loop in efficiency.

8.8.3 Case Statements

Many programming languages include a variant on the case statement. Fortran used the "computed goto." Bcpl and c have a `switch` construct. Pl/i had a generalized construct that mapped well onto a nested set of `if–then–else` statements. As the introduction to this chapter hinted, implementing a case statement efficiently is complex.

Consider c's `switch` statement. The implementation strategy should be:

1. evaluate the controlling expression
2. branch to the selected case
3. execute the code for that case
4. branch to the following statement

Steps 1, 3, and 4 are well understood; they follow from discussions elsewhere in this chapter. The complicated part of implementing a case statement is emitting efficient code to locate the designated case.

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 on the left could be translated into the nest of `if` statements on the right.

```
switch (b×c+d)
{
  case 0:  block$_0$;
           break;
  case 1:  block$_1$;
           break;
  ...
  case 9:  block$_9$;
           break;
  default: block$_{10}$;
           break;
}
```

```
$t_1 \leftarrow b \times c + d$
if ($t_1 = 0$)
   then block$_0$
else if ($t_1 = 1$)
   then block$_1$
else if ($t_1 = 2$)
   then block$_2$
...
else if ($t_1 = 9$)
   then block$_9$
   else block$_{10}$
```

This translation preserves the meaning of the case statement, but makes the cost of reaching individual cases dependent on the order in which they are written. In essence, this code uses linear search to discover the desired case. Still, with a small number of cases, this strategy is reasonable.

Binary Search As the number of individual cases in the case statement rises, the efficiency of linear search becomes a problem. The classic answers to efficient search apply in this situation. 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 search.

The idea is simple. The compiler builds a compact, ordered table of case labels, along with their corresponding branch labels. It uses binary search to discover a matching case label, or the absence of a match. Finally, it branches to the corresponding label.

For the case statement shown above, the following search routine and branch table might be used.

Value	Label
0	$\texttt{LB}_0$
1	$\texttt{LB}_1$
2	$\texttt{LB}_2$
3	$\texttt{LB}_3$
4	$\texttt{LB}_4$
5	$\texttt{LB}_5$
6	$\texttt{LB}_6$
7	$\texttt{LB}_7$
8	$\texttt{LB}_8$
9	$\texttt{LB}_9$

```
$t_1 \leftarrow b \times c + d$
down ← 0
up ← 9
while (down < up)
{
  middle ← (up + down + 1) ÷ 2
  if (Value[middle] ≤ $t_1$)
    then down ← middle
    else up ← middle
}
if (Value[up] = $t_1$)
  then branch Label[up]
  else branch $LB_{10}$
```

The code fragments for each block are now independent. The code fragment for block i begins with a label, $\texttt{LB}_i$, and ends with a branch to $\texttt{L}_{next}$.

The binary search discovers the appropriate case label, if it exists, in $\log_2(n)$ iterations, where n is the number of cases. If the label does not exist, it discovers that fact and branches to the block for the `default` case.

Directly Computing the Address If the case labels form a dense set, the compiler can do better than binary search. In the example, the case statement has labels for every integer from zero to nine. In this situation, the compiler can build a vector that contains the block labels, $\texttt{LB}_i$, and find the appropriate label by performing a simple address calculation.

For the example, the label can be found by computing $\texttt{t}_1$ as before, and using $\texttt{t}_1$ as an index into the table. In this scenario, the code to implement the case statement might be:

```
$t_1 \leftarrow b \times c + d$
if ($0 > t_1$ || $t_1 > 9$)
   then branch to $LB_{10}$
   else
     $t_2 \leftarrow$ memory(@Table + $t_1 \times 4$)
     branch to $t_2$
```

assuming that the representation of a label is four bytes.

With a dense set of labels, this scheme generates efficient code. The cost is both small and constant. 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 compiler can create a block that generates the appropriate run-time error message and use that in place of the default label.

Choosing Between Them The compiler must select an appropriate implementation scheme for each case statement. The decision depends on the number of cases and the properties of the set of case labels. For a handful of cases (≤ 4), the nested `if–then–else` scheme works well. When a larger set of cases exists, but the values do not form a compact set, binary search is a reasonable alternative. (Although, a programmer who steps through the assembly code in a debugger might be rather surprised to find a while loop embedded in the case statement!) When the set is compact, a direct computation using a jump table is probably preferred.

8.8.4 Break Statements

Several languages implement variations on a `break` statement. It appears inside loops and inside case statements. It has the effect of causing execution to continue immediately after the innermost executing control statement. Thus, a `break` inside a loop transfers control to the statement that follows the innermost loop that is currently active. In a case statement, a `break` transfers control to the statement that follows the case statement.

These actions have simple implementations. Each of our loop and case examples ends with a label for the statement that follows the loop. A break would be implemented as an unconditional branch to that label. Notice that a `break` inside a loop implies that the loop body contains more than one basic block. (Otherwise, the break would execute on the first iteration.) Some languages have included a `skip` mechanism that jumps to the next iteration. It can be implemented as a branch to the code that re-evaluates 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.

8.9 Procedure Calls

This section will appear later in the semester.

8.10 Implementing Object-Oriented Languages

This section has yet to be written.

Questions

1. Consider the character copying loop shown on page 234, using explicit string lengths. It uses two `cmp`/`cbr` pairs to implement the end-of-loop tests. In an environment where the size of compiled code is critical, the compiler might replace the `cmp`/`cbr` pair at the end of the loop with a `br` to $\mathtt{L}_1$.

   How would this change affect execution time for the loop? Are there machine models where it runs faster? Are there machine models where it runs slower?

2. Figure 8.8 shows how to use word-oriented memory operations to perform a character string assignment for two word-aligned strings. Arbitrary assignments can generate misaligned cases.

   (a) Write the ILOC code that you would like your compiler to emit for an arbitrary PL/I-style character assignment, such as

   ```
   fee(i:j) = fie(k:l);
   ```

   where `j-i = l-k`. Include versions using character-oriented memory operations and versions using word-oriented memory operations. You may assume that `fee` and `fie` do not overlap in memory.

   (b) The programmer can create character strings that overlap. In PL/I, the programmer might write

   ```
   fee(i:j) = fee(i+1:j+1);
   ```

   or, even more diabolically,

   ```
   fee(i+k:j+k) = fee(i:j);
   ```

   How does this complicate the code that the compiler must generate for the character assignment.

   (c) Are there optimizations that the compiler could apply to the various character-copying loops that would improve run-time behavior? How would they help?

Chapter Notes

Bernstein provides a detailed discussion of the options that arise in generating code for the case statement [11].

Chapter 10

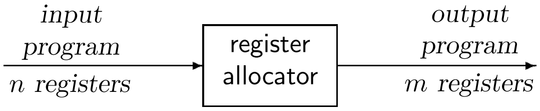
Register Allocation

10.1 The Problem

Registers are the fastest locations in the memory hierarchy. Often, they are the only memory locations that most operations can access directly. The proximity of registers to the functional units makes good use of registers a critical factor in run-time performance. In compiled code, responsibility for making good use of the target machine's register set lies with register allocator.

The register allocator determines, at each point in the program, which values will reside in registers, and which register will hold each such value. When the allocator cannot keep a value in a register throughout its lifetime, the value must be stored in memory and moved between memory and a register on a reference-by-reference basis. The allocator might relegate a value to memory because the code contains more interesting values than the target machine's register set can hold. Alternatively, the value might be confined to memory because the allocator cannot tell if it can safely stay in a register.

Conceptually, the register allocator takes a program that uses some set of registers as its input. It produces as its output an equivalent program that fits into the register set of the target machine.



When the allocator cannot keep some value in a register, it must store the value back to memory and load it again when it is next needed. This process is called *spilling* the value to memory.

Typically, the register allocator's goal is to make effective use of the register set provided by the target machine. This includes minimizing the number of load and store operations that execute to perform spilling. However, other goals are possible. For example, in a memory-constrained environment, the user might want the allocator to minimize the amount of memory needed by the running program.

A bad decision in the register allocator causes some value to be spilled when it might otherwise reside in a register. Because a bad decision leads to extra memory operations, the cost of a misstep by the allocator rises with increasing memory latency. The dramatic increases in memory latency in the 1990s led to a spate of research work on improvements to register allocation.

The remainder of this section lays out the background material needed to discuss the problems that arise in register allocation and the methods that have been used to address them. Subsequent sections present algorithms for allocating registers in a single basic block, across entire procedures, and across regions that span more than a single block but less than the entire procedure.

10.1.1 Memory models

The compiler writer's choice of a memory model (see Section 6.5.2) defines many details of the problem that the register allocator must address.

Register-to-register model Under this model, the compiler treats the IR program as a specification for which values can legally reside in registers. The allocator's task is to map the set of registers used in the input program, called virtual registers, onto the registers provided by the target machine, called physical registers. Register allocation is needed to produce correct code. In this scheme, the allocator inserts additional loads and stores, usually making the compiled code execute more slowly.

Memory-to-memory model Under this model, the compiler trusts the register allocator to determine when it is both safe and profitable to promote values into registers. The IR program keeps all values in memory, moving them in and out of registers as they are used and defined. Typically, the input program uses fewer registers than are available in the target machine. The allocator is an optional transformation that speeds up the code by removing load and store operations.

Under both models, the allocator tries to minimize the number of memory operations executed by the compiled code. However, the allocator must recognize that some values cannot be kept in registers for any non-trivial period of time (see Section 8.2). If the value can be accessed under more than one name, the compiler may be unable to prove that keeping the value in a register is safe. When the compiler cannot determine precisely where a value is referenced, it must store the value in memory, where the addressing hardware will disambiguate the references at run-time (see Sections 10.6.3 and 8.2).

For some values, the compiler can easily discover this knowledge. Examples include scalar local variables and call-by-value parameters, as long as the programmer does not explicitly assign their address to a variable. These values are unambiguous. Other values require that the compiler perform extensive analysis to determine whether or not it can keep the value in a register. Examples include some references to array elements, pointer-based values, and call-by-reference

```
main() {
  int *A[], *B[], i, j;
  int C[100], D[100];
  if (fee()) {
    A = C;
    B = D;
  }
  else {
    A = C;
    B = C;
  }
  j = fie();
  for (i=0; i<100; i++)
    A[i] = A[i] * B[j];
}
```

Within a procedure

```
subroutine fum
  integer x, y
  ...
  call foe(x,x)
  ...
  call foe(x,y)
  end


subroutine foe(a,b)
  integer a, b
  ...
  end
```

Using parameters

Figure 10.1: Examples of ambiguity

formal parameters. These values are considered ambiguous, unless analysis can prove otherwise. Unfortunately, even the best analysis cannot disambiguate all memory references.

Programmers can easily write programs that defy analysis. The simplest example uses a common code sequence in multiple contexts—some ambiguous, others unambiguous. Consider the two examples shown in Figure 10.1. The C code on the left creates two different contexts for the loop by performing a pointer assignment in each side of the `if–then–else` construct. (The details of `fee()` and `fie()` are irrelevant for this example.) If `fee` returns true, then `A` and `B` point to different memory locations inside the loop and `B[j]` can be kept in a register. Along the other path, `A` and `B` point to the same storage locations. If $0 \leq$ `j` < 100, then `B[j]` cannot be kept in a register.

The example on the right, in FORTRAN, creates the same effect using call-by-reference parameters. The first call to `foe` creates an environment where `foe`'s parameters `a` and `b` refer to the same storage location. The second call creates an environment where `a` and `b` refer to distinct storage locations. Unless the compiler radically transforms the code, using techniques such as inline substitution or procedure cloning (see Chapter 14), it must compile a single executable code sequence that functions correctly in both these environments. The compiler cannot keep either `a` or `b` in a register, unless it proves that every invocation of `foe` occurs in an environment where they cannot occupy the same storage location, or it proves that `b` is not referenced while `a` is in a register, and *vice-versa*. (See Chapter 13).

For complex access patterns, the compiler may not know whether two distinct names refer to the same storage location. In this case, the compiler cannot keep either value in a register across a definition of the other. In practice, the compiler must behave conservatively, by leaving ambiguous values in memory, and moving them into registers for short periods when they are defined or used.

Thus, lack of knowledge can keep the compiler from allocating a variable to a register. This can result from limitations in the compiler's analysis. It can also occur when a single code sequence inherits different environments along different paths. These limitations in what the compiler can know tend to favor the register-to-register model. The register-to-register model provides a mechanism for other parts of the compiler to encode knowledge about ambiguity and uniqueness. This knowledge might come from analysis; it might come from understanding the translation of a complex construct; or it might be derived from the source text in the parser.

10.1.2 Allocation versus Assignment

In a modern compiler, the register allocator solves two distinct problems—register allocation and register assignment. These problems are related but distinct.

Allocation Register allocation maps an unlimited set of names onto the finite set of resources provided by the target machine. In a register-to-register model, it maps virtual registers onto a new set of names that models the physical register set, and spills any values that do not fit in the register set. In a memory-to-memory model, it maps some subset of the memory locations onto a set of names that models the physical register set. Allocation ensures that the code will map onto the target machine's register set, at each instruction.

Assignment Register assignment maps an allocated name space onto the physical registers of the target machine. It assumes that the allocation has been performed, so that code will fit into the set of the physical registers provided by the target machine. Thus, at each instruction in the generated code, no more than k values are designated as residing in registers, where k is the number of physical registers. Assignment produces the actual register names required in the executable code.

Register allocation is, in almost any realistic example, NP-Complete. For a single basic block, with one size of data value, optimal allocation can be done in polynomial time, as long as the cost of storing values back to memory is uniform. Almost any additional complexity in the problem makes it NP-Complete. For example, add a second size of data item, such as a register pair that holds a double-precision floating point number, and the problem becomes NP-Complete. Alternatively, add a realistic memory model, or the fact that some values need not be stored back to memory, and the problem becomes NP-Complete. Extend the scope of allocation to include control flow and multiple blocks, and the problem becomes NP-Complete. In practice, one or more of these problems arise in compiling for almost any real system.

Register assignment, in many cases, can be solved in polynomial time. Given a feasible allocation for a basic block—that is, one where demand for physical registers at each instruction does not exceed the number of physical registers—an assignment can be produced in linear time using an analogy to interval graph coloring. The related problem on an entire procedure can be solved in polynomial time—that is, if, at each instruction, demand for physical registers does not exceed the number of physical registers, then a polynomial time algorithm exists for deriving an assignment.

The distinction between allocation and assignment is both subtle and important. It is often blurred in the literature and in implementation. As we shall see, most "register allocators" perform both functions, often at the same time.

10.1.3 Register Classes

The physical registers provided by most processors do not form a homogenous pool of interchangeable resources. Typical processors have distinct classes of registers for different kinds of values.

For example, most modern computers have both *general purpose registers* and *floating-point registers*. The former hold integer values and memory addresses, while the latter hold floating-point values. This dichotomy is not new; the early Ibm 360 machines had 16 general-purpose registers and 4 floating-point registers. Modern processors may add more classes. For example, the Ibm/Motorola PowerPC has a separate register class for condition codes, and the Intel ia-64 has separate classes for predicate registers and branch target registers. The compiler must place each value in the appropriate register class. The instruction set enforces these rules. For example, a floating-point multiply operation can only take arguments from the floating-point register set.

If the interactions between two register classes are limited, the compiler may be able to solve the problems independently. This breaks the allocation problem into smaller, independent components, reduces the size of the data structures and may produce faster compile times. When two register classes overlap, however, then both classes must be modeled in a single allocation problem. The common architectural practice of keeping double-precision floating-point numbers in pairs of single-precision registers is a good example of this effect. The class of paired, or double-precision registers and the class of singleton, or single-precision registers both map onto the same underlying set of hardware registers. The compiler cannot allocate one of these classes without considering the other, so it must solve the joint allocation problem.

Even if the different register classes are physically and logically separate, they interact through operations that refer to registers in multiple classes. For example, on many architectures, the decision to spill a floating-point register requires the insertion of an address calculation and some memory operations; these actions use general-purpose registers and change the allocation problem for general-purpose registers. Thus, the compiler can make independent allocation decisions for the different classes, but those decisions can have consequences that affect the allocation in other register classes. Spilling a predicate register or a condition-code register has similar effects. This suggests that general-purpose register allocation should occur after the other register classes.

10.2 Local Register Allocation and Assignment

As an introduction to register allocation, consider the problems that arise in producing a good allocation for a single basic block. In optimization, methods that handle a single basic block are termed *local* methods, so these algorithms are local register-allocation techniques. The allocator takes as input a single basic block that incorporates a register-to-register memory model.

To simplify the discussion, we assume that the program starts and ends with the block; it inherits no values from blocks that executed earlier and leaves behind no values for blocks that execute later. Both the target machine and the input code use a single class of registers. The target machine has k registers.

The code shape encodes information about which values can legally reside in a register for non-trivial amounts of time. Any value that can legally reside in a register is kept in a register. The code uses as many register names as needed to encode this information, so it may name more registers that the target machine has. For this reason, we call these pre-allocation registers *virtual register.* For a given block, the number of virtual registers that it uses is `MaxVR`.

The basic block consists of a series of N three-address operations op_1, op_2, op_3, …, op_N. Each operation has the form $\texttt{op}_i\ \texttt{vr}_{i_1}\texttt{,vr}_{i_2} \Rightarrow \texttt{vr}_{i_3}$. The notation `vr` denotes the fact that these are virtual registers, rather than physical registers. From a high-level view, the goal of local register allocation is to create an equivalent block where each reference to a virtual register is replaced with a reference to a specific physical register. If $\texttt{MaxVR} > k$, the allocator may need to insert loads and stores, as appropriate, to fit the code into the set of k physical registers. An alternative statement of this property is that the output code can have no more than k values in a register at any point in the block.

We will explore two approaches to this problem. The first approach counts the number of references to a value in the block and uses these frequency counts to determine which values will reside in registers. Because it relies on externally-derived information—the frequency counts— to make its decisions, we consider this approach a top-down approach. The second approach relies on detailed, low-level knowledge of the code to make its decisions. It walks over the block and computes, at each operation, where or not a spill is needed. Because it synthesizes and combines many low-level facts to drive its decision-making process, we consider this a bottom-up approach.

10.2.1 Top-down Local Register Allocation

The top-down local allocator works from a simple principle: the most heavily used values should reside in registers. To implement this heuristic, it counts the number of occurrences of each virtual register in the block, and uses these frequency counts as priorities to allocate virtual registers to physical registers.

If there are more virtual registers than physical registers, the allocator must reserve several physical registers for use in computations that involve values allocated to memory. The allocator must have enough registers to address and load two operands, to perform the operation, and to store the result. The precise number of registers depends on the target architecture; on a typical RISC machine, the number might be two to four registers. We will refer to this machine-specific number as "*feasible*."

To perform top-down local allocation, the compiler can apply the following simple algorithm.

(1) Compute a score to rank each virtual register by counting all the uses of the virtual register. This takes a linear walk over the operations in the block; it increments `score[`VR_i`]` each time it finds VR_i in the block.

(2) Sort the VRs into rank order. If blocks are reasonably small, it can use a radix sort, since `score[`VR_i`]` is bound by a small multiple of the block length.

(3) Assign registers in priority order. The first $k - \textit{feasible}$ virtual registers are assigned physical registers.

(4) Rewrite the code. Walk the code a second time, rewriting it to reflect the new allocation and assignment. Any VR assigned a physical register is replaced with the name of that physical register. Any VR that did not receive a register uses one of the registers reserved for temporary use. It is loaded before each use and stored after each definition.

The strength of this approach is that it keeps heavily used virtual registers in physical registers. Its primary weakness lies in the approach to allocation—it dedicates a physical register to the virtual register for the entire basic block. Thus, a value that is heavily used in the first half of the block and unused in the second half of the block occupies the physical register through the second half, even though it is no longer of use. The next section presents a technique that addresses this problem. It takes a fundamentally different approach to allocation—a bottom-up, incremental approach.

10.2.2 Bottom-up Local Allocation

The key idea behind the bottom-up local allocator is to focus on the transitions that occur as each operation executes. It begins with all the registers unoccupied. For each operation, the allocator needs to ensure that its operands are in registers before it executes. It must also allocate a register for the operation's result. Figure 10.2 shows the basic structure of a local, bottom-up allocator.

The bottom-up allocator iterates over the operations in the block, making allocation decisions on demand. There are, however, some subtleties. By considering $\texttt{vr}_{i_1}$ and $\texttt{vr}_{i_2}$ in order, the allocator avoids using two physical registers for an operation with a repeated operand, such as `add` $\texttt{r}_y,\texttt{r}_y \Rightarrow \texttt{r}_z$. Similarly, trying to free $\texttt{r}_x$ and $\texttt{r}_y$ before allocating $\texttt{r}_z$ avoids spilling a register to hold the result when the operation actually frees up a register. All of the complications are hidden in the routines *ensure, allocate* and *free*.

The routine *ensure* is conceptually simple. In pseudo-code, it looks like:

```
for each operation, $i$, in order 1 to $N$
  $\texttt{r}_x \leftarrow$ ensure($\texttt{vr}_{i_1}$, class($\texttt{vr}_{i_1}$))
  $\texttt{r}_y \leftarrow$ ensure($\texttt{vr}_{i_2}$, class($\texttt{vr}_{i_2}$))
  if $r_x$ is not needed after $i$ then
      free($\texttt{r}_x$,class($\texttt{r}_x$))
  if $r_y$ is not needed after $i$ then
      free($\texttt{r}_y$,class($\texttt{r}_y$))
  $\texttt{r}_z \leftarrow$allocate($\texttt{vr}_{i_3}$, class($\texttt{vr}_{i_3}$))
  emit op$_i$ $\texttt{r}_x, \texttt{r}_y \Rightarrow \texttt{r}_z$
```

Figure 10.2: The bottom-up, local register allocator

```
ensure(vr,class)
  if (vr is already in class) then
      result $\leftarrow$ physical register holding vr
  else
      result $\leftarrow$ allocate(vr,class)
      emit code to move vr $\Rightarrow$ result
  return result
```

It takes two arguments, a virtual register holding the desired value, and a representation for the appropriate register class, `class`. If the virtual register already occupies a physical register, *ensure*'s job is done. Otherwise, it allocates a physical register for the virtual register and emits code to move the virtual register into that physical register. In either case, it returns the physical register.

Allocate and *free* expose the details of the problem. Understanding their actions requires more information about the representation of a register class. The class contains information on each physical register in the class. In particular, at each point in the allocation, the class holds: a flag indicating whether the physical register is allocated or free, the name of the virtual register, if any, that it holds, and the index in the block of that virtual register's next reference. To make this efficient, it also needs a list of unallocated (or free) registers. The implementation of `class` contains a stack for this purpose. Figure 10.3 shows this might be declared in C. The routine on the right side of the figure shows how the structure should be initialized.

With this level of detail, implementing both *allocate* and *free* is straightforward.

```
struct Class {
  int Size;
  int Name[Size];
  int Next[Size];
  int Free[Size];
  int Stack[Size];
  int StackTop;
}
```

```
initialize(class,size)
  class.Size $\leftarrow$ size
  class.StackTop $\leftarrow$ -1
  for i $\leftarrow$ 1 to size-1
     class.Name[i] $\leftarrow \perp$
     class.Next[i] $\leftarrow \infty$
     class.Free[i] $\leftarrow$ true
     push(i,class)
```

Figure 10.3: Representing a register class in C

```
allocate(vr,class)
  if (class.StackTop $\geq$ 0)
     i $\leftarrow$ pop(class)
  else
     i $\leftarrow$ j that maximizes
        class.Next[j]
     store contents of i
  class.Name[i] $\leftarrow$ vr
  class.Next[i] $\leftarrow$ dist(vr)
  class.Free[i] $\leftarrow$ false
  return i
```

```
free(i,class)
  if (class.Free[i] $\neq$ true)
     push(i,class)
     class.Name[i] $\leftarrow \perp$
     class.Next[i] $\leftarrow \infty$
     class.Free[i] $\leftarrow$ true
```

Each class maintains a list of free physical registers, in stack form. `Allocate` returns a physical register from the free list of `class`, if one exists. Otherwise, it selects the value stored in `class` that is used farthest in the future, stores it, and re-allocates the physical register for *vr*. `Free` pushes the register onto the stack and resets its fields in the `class` structure.

The function *dist*(`vr`) returns the index in the block of the next reference to `vr`. The compiler can annotate each reference in the block with the appropriate *dist* value by making a single backward pass over the block.

The net effect of this bottom-up technique is straightforward. Initially, it assumes that the physical registers are unoccupied and places them on a free list. For the first few operations, it satisfies demand from the free list. When the allocator needs another register and discovers that the free list is empty, it must spill some existing value from a register to memory. It picks the value whose next use is farthest in the future. As long as the cost of spilling a value is the same for all the registers, then this frees up the register for the longest period of time. In some sense, it maximizes the benefit obtained for the cost of the spill. This algorithm is quite old, it was first proposed by Sheldon Best for the original Fortran compiler in the mid-1950s.

This algorithm produces excellent local allocations. Several authors have argued that it produces optimal allocations. Complications that arise in practice make the argument for optimality tenuous. At any point in the allocation, some values in registers may need to be stored on a spill, while others may not. For example, if the register contains a known constant value, the store is superfluous since the allocator can recreate the value without a copy in memory. Similarly, a value that was created by a load from memory need not be stored. A value that need not be stored is considered *clean*, while a value that needs a store is *dirty*. A version of the bottom-up local allocator that first spills the furthest clean value, and, if no clean value remains, then spills the furthest dirty value, will produce excellent local allocations—better than the top-down allocator described above.

The bottom-up local allocator differs from the top-down local allocator in the way that it handles individual values. The top-down allocator devotes a physical register to a virtual register for the entire block. The bottom-up allocator assigns a physical register to a virtual register for the distance between two consecutive references to the virtual register. It reconsiders that decision at each invocation of `allocate`—that is, each time that it needs another register. Thus, the bottom-up algorithm can, and does, produce allocations where a single virtual register is kept in different locations at different points in its lifetime. Similar behavior can be retrofitted into the top-down allocator. (See question 1 at the end of the chapter.)

10.3 Moving beyond single blocks

We have seen how to build good allocators for single blocks. Working top down, we arrived at the frequency count allocator. Working bottom up, we arrived at Best's allocator. Using the lessons from Best's allocator, we can improve the frequency count allocator. The next step is to extend the scope of allocation beyond single basic blocks.

Unfortunately, moving from a single block to multiple blocks invalidates many of the assumptions that underlie the local allocation schemes. For example, with multiple blocks, the allocator can no longer assume that values do not flow between blocks. The entire purpose of moving to a large scope for allocation is to account for the fact that values flow between blocks and to generate allocations that handle such flow efficiently. The allocator must correctly handle values computed in previous blocks, and it must preserve values for uses in later blocks. To accomplish this, the allocator needs a more sophisticated way of handling "values" than the local allocators use.

10.3.1 Liveness and Live Ranges

Regional and global allocators try to assign values to registers in a way that coordinates their use across multiple blocks. To accomplish this, the allocators compute a new name space that reflects the actual patterns of definition and use for each value. Rather than considering variables or values, the allocator works from a basis of *live ranges*. A single live range consists of a set of definitions and uses that are related to each other because their values flow together. That is, a live range contains a set of definitions and a set of uses. This set is self-contained, in the sense that every definition that can reach a use is in the same live range. Symmetrically, every use that a definition can reach is in the same

```
 1. loadI   @stack    ⇒ r_arp
 2. loadAI  r_arp, 0  ⇒ r_w
 3. loadI   2         ⇒ r_2
 4. loadAI  r_arp, 8  ⇒ r_x
 5. loadAI  r_arp, 16 ⇒ r_y
 6. loadAI  r_arp, 24 ⇒ r_z
 7. mult    r_w, r_2  ⇒ r_w
 8. mult    r_w, r_x  ⇒ r_w
 9. mult    r_w, r_y  ⇒ r_w
10. mult    r_w, r_z  ⇒ r_w
11. storeAI r_w       ⇒ r_arp, 0
```

Live	*Register*	
Range	*Name*	*Interval*
1	r_{arp}	[1,11]
2	r_w	[2,7]
3	r_w	[7,8]
4	r_w	[8,9]
5	r_w	[9,10]
6	r_w	[10,11]
7	r_2	[3,7]
8	r_x	[4,8]
9	r_y	[5,9]
10	r_z	[6,10]

Figure 10.4: Live ranges in a basic block

live range as the definition.

The term "live range" relies, implicitly, on the notion of *liveness*—one of the fundamental ideas in compile-time analysis of programs.

> At some point p in a procedure, a value v is *live* if it has been defined along a path from the procedure's entry to p and a path along which v is not redefined exists from p to a use of v

Thus, if v is live at p, then v must be preserved because subsequent execution may use v. The definition is carefully worded. A path exists from p to a use of v. This does not guarantee that any particular execution will follow the path, or that any execution will ever follow the path. The existence of such a path, however, forces the compiler to preserve v for the potential use.

The set of live ranges is distinct from the set of variables or values. Every value computed in the code is part of some live range, even if it has no name in the original source code. Thus, the intermediate results produced by address computations have live ranges, just the same as programmer-named variables, array elements, and addresses loaded for use as a branch target. A specific programmer-named variable may have many distinct live ranges. A register allocator that uses live ranges can place those distinct live ranges in different registers. Thus, a variable, x, might reside in different registers at two distinct points in the executing program.

To make these ideas concrete, consider the problem of finding live ranges in a single basic block. Figure 10.4 shows the block from Figure 1.1, with an initial operation added to initialize r_{arp}. All other references to r_{arp} inside the block are uses rather than definitions. Thus, a single value for r_{arp} is used throughout the block. The interval $[1, 11]$ represents this live range. Consider r_w. Operation 1 defines r_w; operation 6 uses that value. Operations 6, 7, 8, and 9 each define a new value stored in r_w; in each case, the following operation uses the value. Thus, the register named r_w in the figure holds a number of distinct live ranges—specifically $[2, 7]$, $[7, 8]$, $[8, 9]$, $[9, 10]$, and $[10, 1]$. A register

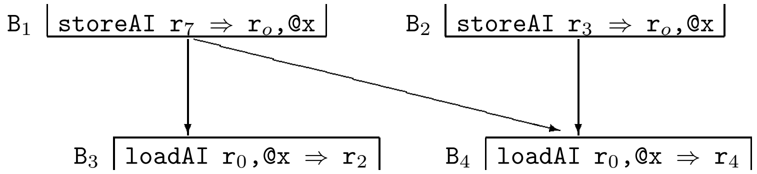


Figure 10.5: Problems with multiple blocks

allocator need not keep these distinct live ranges of $\texttt{r}_w$ in the same register. Instead, each live range in the block can be treated as an independent value for allocation and assignment. The table on the right side of Figure 10.4 shows all of the live ranges in the block.

To find live ranges in regions larger than a single block, the compiler must discover the set of values that are live on entry to each block, as well as those that are live on exit from each block. To summarize this information, the compiler can annotate each basic block b with sets LiveIn(b) and LiveOut(b)

LiveIn A value $x \in$ LiveIn(b) if and only if it is defined along some path through the control-flow graph that leads to b and it is either used directly in b, or is in LiveOut(b). That is, $x \in$ LiveIn(b) implies that x is live just before the first operation in b.

LiveOut A value $x \in$ LiveOut(b) if and only if it is used along some path leaving b, and it is either defined in b or is in LiveIn(b). That is, x is live immediately after the last operation in b.

Chapter 13 shows how to compute LiveIn and LiveOut sets for each block. At any point p in the code, values that are not live need no register. Similarly, the only values that need registers at point p are those values that are live at p, or some subset of those values. Local register allocators, when implemented in real compilers, use Live sets to determine when a value must be preserved in memory beyond its last use in the block. Global allocators use analogous information to discover live ranges and to guide the allocation process.

10.3.2 Complications at Block Boundaries

A compiler that uses local register allocation might compute LiveIn and LiveOut sets for each block as a necessary prelude to provide the local allocator with information about the status of values at the block's entry and its exit. The presence of these sets can simplify the task of making the allocations for individual blocks behave appropriately when control flows from one block to another. For example, a value in LiveOut(b) must be stored back to memory after a definition in b; this ensures that the value will be in the expected location when it is loaded in a subsequent block. In contrast, if the value is not in LiveOut(b), it need not be stored, except as a spill for later use inside b.

Some of the effects introduced by considering multiple blocks complicate either assignment or allocation. Figure 10.5 suggests some of the complications that arise in global assignment. Consider the transition that occurs along the edge from block $\mathtt{B}_1$ to block $\mathtt{B}_3$.

$\mathtt{B}_1$ has the value of program variable `x` in $\mathtt{r}_7$. $\mathtt{B}_3$ wants it in $\mathtt{r}_2$. When it processes $\mathtt{B}_1$, the allocator has no knowledge of the context created by the other blocks, so it must store `x` back to `x`'s location in memory (at offset `@x` from the ARP in $\mathtt{r}_0$). Similarly, when the allocator processes $\mathtt{B}_3$, it has no knowledge about the behavior of $\mathtt{B}_1$, so it must load `x` from memory. Of course, if it knew the results of allocation on $\mathtt{B}_1$, it could assign `x` to $\mathtt{r}_7$ and make the load unnecessary. In the absence of this knowledge, it must generate the load. The references to `x` in $\mathtt{B}_2$ and $\mathtt{B}_4$ further complicate the problem. Any attempt to coordinate `x`'s assignment across blocks must consider both those blocks since $\mathtt{B}_4$ is a successor of $\mathtt{B}_1$, and any change in $\mathtt{B}_4$'s treatment of `x` has an impact in its other predecessor, $\mathtt{B}_2$.

Similar effects arise with allocation. What if `x` were not referenced in $\mathtt{B}_2$? Even if we could coordinate assignment globally, to ensure that `x` was always in $\mathtt{r}_7$ when it was used, the allocator would need to insert a load of `x` at the end of $\mathtt{B}_2$ to let $\mathtt{B}_4$ avoid the initial load of `x`. Of course, if $\mathtt{B}_2$ had other successors, they might not reference `x` and might need another value in $\mathtt{r}_7$.

These fringe effects at block boundaries can become complex. They do not fit into the local allocators because they deal with phenomena that are entirely outside its scope. If the allocator manages to insert a few extra instructions that iron out the differences, it may choose to insert them in the wrong block—for example, in a block that forms the body of an inner loop rather than in that loop's header block. The local models assume that all instructions execute with the same frequency; stretching the models to handle larger regions invalidates that assumption, too. The difference between a good allocation and a poor one may be a few instructions in the most heavily executed block in the code.

A second issue, more subtle but more problematic, arises when we try to stretch the local allocation paradigms beyond single blocks. Consider using Best's algorithm on block $\mathtt{B}_1$. With only one block, the notion of the "furthest" next reference is clear. The local algorithm has a unique distance to each next reference. With multiple successor blocks, the allocator must choose between references along different paths. For the last reference to some value `y` in $\mathtt{B}_1$, the next reference is either the first reference to `y` in $\mathtt{B}_3$ or the first reference to `y` in $\mathtt{B}_4$. These two references are unlikely to be in the same position, relative to the end of $\mathtt{B}_1$. Alternatively, $\mathtt{B}_3$ might not contain a reference to `y`, while $\mathtt{B}_4$ does. Even if both blocks use `y`, and the references are equidistant in the input code, local spilling in one block might make them different in unpredictable ways. The basic premise of the bottom-up local method begins to crumble in the presence of multiple control-flow paths.

All of these problems suggest that a different approach is needed to move beyond local allocation to regional or global allocation. Indeed, the successful global allocation algorithms bear little resemblance to the local algorithms.

10.4 Global Register Allocation and Assignment

The register allocator's goal is to minimize the execution time required for instructions that it must insert. This is a global issue, not a local one. From the perspective of execution time, the difference between two different allocations for the same basic code lies in the number of loads, stores, and copy operations inserted by the allocator and their placement in the code. Since different blocks execute different numbers of times, the placement of spills has a strong impact on the amount of execution time spent in spill code. Since block execution frequencies can vary from run to run, the notion of a best allocation is somewhat tenuous—it must be conditioned to a particular set of block execution frequencies.

Global register allocation differs from local allocation in two fundamental ways.

1. The structure of a live range can be more complex than in the local allocator. In a single block, a live range is just an interval in a linear string of operations. Globally, a live range is the set of definitions that can reach a given use, along with all the uses that those definitions can reach. Finding live ranges is more complex in a global allocator.

2. Distinct references to the same variable can execute a different number of times. In a single block, if any operation executes, all the operations execute (unless an exception occurs), so the cost of spilling is uniform. In a larger scope, each reference can be in a different block, so the cost of spilling depends on where the references are found. When it must spill, the global allocator should consider the spill cost of each live range that is a candidate to spill.

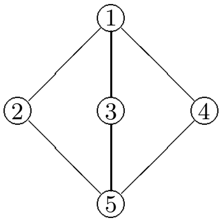
Any global allocator must address both these issues. This makes global allocation substantially more complex than local allocation.

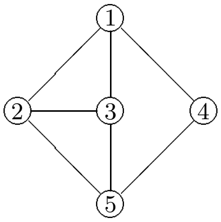
To address the issue of complex live ranges, global allocators explicitly create a name space where each distinct live range has a unique name. Thus, the allocator maps a live range onto either a physical register or a memory location. To accomplish this, the global allocator first constructs live ranges and renames all the virtual register references in the code to reflect the new name space constructed around the live ranges. To address the issue of execution frequencies, the allocator can annotate each reference or each basic block with an estimated execution frequency. The estimates can come from static analysis or from profile data gathered during actual executions of the program. These estimated execution frequencies will be used later in the allocator to guide decisions about allocation and spilling.

Finally, global allocators must make decisions about allocation and assignment. They must decide when two values can share a single register, and they must modify the code to map each such value to a specific register. To accomplish these tasks, the allocator needs a model that tells it when two values can (and cannot) share a single register. It also needs an algorithm that can use the

Digression: Graph Coloring

Many global register allocators use *graph coloring* as a paradigm to model the underlying allocation problem. For an arbitrary graph G, a coloring of G assigns a color to each node in G so that no pair of adjacent nodes have the same color. A coloring that uses k colors is termed a k coloring, and k is the graph's *chromatic number*. Consider the following graphs:





The graph on the left is 2-colorable. For example, assigning *blue* to nodes 1 and 5, and *red* to nodes 2, 3, and 4 produces the desired result. Adding one edge, as shown on the right, makes the graph 3-colorable. (Assign *blue* to nodes 1 and 5, *red* to nodes 2 and 4, and *white* to node 3.) No 2-coloring exists for the right-hand graph.

For a given graph, the problem of finding its minimal chromatic number is NP-Complete. Similarly, the problem of determining if a graph is k-colorable, for some fixed k, is NP-Complete. Algorithms that use graph-coloring as a paradigm for allocating finite resources use approximate methods that try to discover colorings into the set of available resources.

model to derive effective and efficient allocations. Many global allocators operate on a graph-coloring paradigm. They build a graph to model the conflicts between registers and attempt to find an appropriate coloring for the graph. The allocators that we discuss in this section all operate within this paradigm.

10.4.1 Discovering Global Live Ranges

To construct live ranges, the compiler must discover the relationships that exist between different definitions and uses. The allocator must derive a name space that groups together all the definitions that reach a single use and all the uses that a single definition can reach. This suggests an approach where the compiler assigns each definition a unique name and merges definition names together when they reach a common use.

The static single assignment form (SSA) of the code provides a natural starting point for this construction. Recall, from Section 6.3.6, that SSA assigns a unique name to each definition and inserts ϕ-functions to ensure that each use refers to only one definition. The ϕ-functions concisely record the fact that distinct definitions on different paths in the control-flow graph reach a single reference. Two definitions that flow into a ϕ-function are belong in the same live range because the ϕ-function creates a name representing both values. Any operation that references the name created by the ϕ-function uses one of these values; the specific value depends on how control-flow reached the ϕ-function. Because the two definitions can be referenced in the same use, they belong in the same register. Thus, ϕ-functions are the key to building live ranges.

To build live ranges from SSA, the allocator uses the disjoint-set union-find algorithm [27] and makes a single pass over the code. First, the allocator assigns a distinct set to each SSA name, or definition. Next, it examines each ϕ-function in the program, and unions together the sets of each ϕ-function parameter. After all the ϕ-functions have been processed, the resulting sets represent the maximal live ranges of the code. At this point, the allocator can rewrite the code to use the live range names. (Alternatively, it can maintain a mapping between SSA names and live-range names, and add a level of indirection in its subsequent manipulations.)

10.4.2 Estimating Global Spill Costs

To let it make informed spill decisions, the global allocator must estimate the costs of spilling each value. The value might be a single reference, or it might be an entire live range. The cost of a spill has three components: the address computation, the memory operation, and an estimated execution frequency.

The compiler can choose the spill location in a way that minimizes the cost of addressing; usually, this means keeping spilled values in the procedure's activation record. In this scenario, the compiler can generate an operation such as `loadAI` or `storeAI` for the spill. As long as the ARP is in a register, the spill should not require additional registers for the address computation.

The cost of the memory operation is, in general, unavoidable. If the target machine has local (*i.e.,* on-chip) memory that is not cached, the compiler might use that memory for spilling. More typically, the compiler needs to save the value in the computer's main memory and to restore it from that memory when a later operation needs the value. This entails the full cost of a load or store operation. As memory latencies rise, the cost of these operations grows. To make matters somewhat worse, the allocator only inserts spill operations when it absolutely needs a register. Thus, many spill operations occur in code where demand for registers is high. This may keep the scheduler from moving those operations far enough to hide the memory latency. The compiler must hope that spill locations stay in cache. (Paradoxically, those locations only stay in the cache if they are accessed often enough to avoid replacement—suggesting that the code is executing too many spills.)

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. (Such a live range can result from transformations intended to improve the code; for example, if the use were optimized away and the store resulted from a procedure call rather than the definition of a new value.) Any live range with a negative spill cost should be spilled, since doing so decreases demand for registers and removes instructions from the code.

Infinite Spill Costs Some live ranges are short enough that spilling them never helps the allocation. Consider a use that immediately follows its definition. Spilling the definition and use produces two short live ranges. The first contains the definition followed by a store; the second live range contains a load followed by the use. Neither of these new live ranges uses fewer registers than the original live range, so the spill produced no benefit. The allocator should assign the original live range a spill cost of infinity.

In general, a live range should have infinite spill cost if no interfering live range ends between its definitions and its uses, and no more than $k - 1$ values are defined between the definitions and the uses. The first condition stipulates that availability of registers does not change between the definitions and uses. The second avoids a pathological situation that can arise from a series of spilled copies—m loads followed by m stores, where $m \gg k$. This can create a set of more than k mutually interfering live ranges; if the allocator assigns them all infinite spill costs, it will be unable to resolve the situation.

Accounting for Execution Frequencies To account for the different execution frequencies of the basic blocks in the control-flow graph, the compiler must annotate each block (if not each reference) with an estimated execution count. Most compilers use simple heuristics to estimate execution costs. A common method is to assume that each loop executes ten times. Thus, it assigns a count of ten to a load inside one loop, and a count of one hundred to a load inside two loops. An unpredictable `if-then-else` might decrease the execution count by half. In practice, these estimates ensure a bias toward spilling in outer loops rather than inner loops.

To estimate the spill cost for a single reference, the allocator forms the product

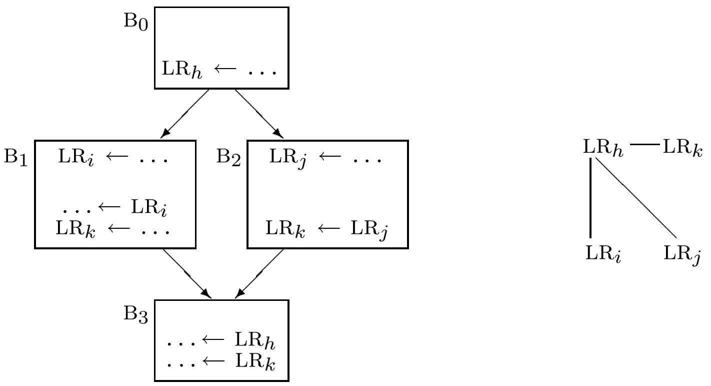
$$(\textit{addressing cost} + \textit{cost of memory operation}) \times \textit{execution frequency}.$$

For each live range, it sums the costs of the individual references. This requires a pass over all the blocks in the code. The allocator can pre-compute these costs for all live ranges, or it can wait until it discovers that it must spill a value.

10.4.3 Interferences and the Interference Graph

The fundamental effect that a global register allocator must model is the competition between values for space in the target machine's register set. Consider two live ranges, LR_i and LR_j. If there exists an operation where both LR_i and LR_j are live, then they cannot reside in the same register. (In general, a physical register can hold only one value.) We say that LR_i and LR_j *interfere*.

To model the allocation problem, the compiler can build an *interference graph*, I. Nodes in I represent individual live ranges. Edges in I represent interferences. Thus, an edge $\langle n_i, n_j \rangle \in I$ exists if and only if the corresponding live ranges, LR_i and LR_j are both live at some operation. The left side of Figure 10.6 shows a code fragment that defines four live ranges, LR_h, LR_i, LR_j, and LR_k. The right side shows the corresponding interference graph. LR_h interferes



Code fragment *Interference graph*

Figure 10.6: Live ranges and interference

with each of the other live ranges. The rest of the live ranges, however, do not interfere with each other.

If the compiler can construct a k-coloring for I, where $k \leq$ the size of the target machine's register set, then it can map the colors directly onto physical registers to produce a legal allocation. In the example, LR_h receives its own color because it interferes with the other live ranges. The other live ranges can all share a single color. Thus, the graph is 2-colorable and the code fragment, as shown, can be rewritten to use just two registers.

Consider what would happen if another phase of the compiler reordered the two operations at the end of B_1. This makes LR_k and LR_i simultaneously live. Since they now interfere, the allocator must add the edge $\langle \text{LR}_k, \text{LR}_i \rangle$ to I. The resulting graph is not 2-colorable. The graph is small enough to prove this by enumeration. To handle this graph, the allocator has two options: use three colors (registers), or, if the target machine has only two registers, to spill one of LR_i or LR_h before the definition of LR_k in B_1. Of course, the allocator could also reorder the two operations and eliminate the interference between LR_i and LR_k. Typically, register allocators do not reorder operations to eliminate interferences. Instead, allocators assume a fixed order of operations and leave ordering questions to the instruction scheduler (see Chapter 11).

Building the Interference Graph Once the allocator has discovered global live ranges and annotated each basic block in the code with its LiveOut set, it can construct the interference graph by making a simple linear pass over each block. Figure 10.7 shows the basic algorithm. The compiler uses the block's LiveOut set as an initial value for LiveNow and works its way backward through the block, updating LiveNow to reflect the operations already processed. At each operation, it adds an edge from the live range being defined to each live range

```
for each LR, $i$
  create a node $n_i \in N$

for each basic block $b$
  LiveNow($b$) ← LiveOut($b$)
  for $op_n$, $op_{n-1}$, $op_{n-2}$, ... $op_1$ in $b$
    with form $op_i$ $LR_j$,$LR_k$ ⇒ $LR_l$
    for each $LR_i$ in LiveNow($b$),
      add $\langle LR_l, LR_i \rangle$ to $E$
    remove $LR_l$ from LiveNow($b$)
    add $LR_j$ & $LR_k$ to LiveNow($b$)
```

Figure 10.7: Constructing the Interference Graph

in LiveNow. It then incrementally updates LiveNow and moves up the block by an instruction.

This method of computing interferences takes time proportional to the size of the LiveNow sets at each operation. The naive algorithm would add edges between each pair of values in LiveNow at each operation; that would require time proportional to the square of the set sizes at each operation. The naive algorithm also introduces interferences between inputs to an operation and its output. This creates an implicit assumption that each value is live beyond its last use, and prevents the allocator from using the same register for an input and an output in the same operation.

Notice that a copy operation, such as `i2i` $LR_i \Rightarrow LR_j$, does not create an interference between LR_i and LR_j. In fact, LR_i and LR_j may occupy the same physical register, unless subsequent context creates an interference. Thus, a copy that occurs as the last use of some live range can often be eliminated by combining, or coalescing, the two live ranges (see Section 10.4.6).

To improve efficiency later in the allocator, several authors recommend building two representations for I, a lower-diagonal bit-matrix and a set of adjacency lists. The bit matrix allows a constant time test for interference, while the adjacency lists make iterating over a node's neighbors efficient. The bit matrix might be replaced with a hash table; studies have shown that this can produce space savings for sufficiently large interference graphs. The compiler writer may also treat disjoint register classes as separate allocation problems to reduce both the size of I and the overall allocation time.

Building an Allocator To build a global allocator based on the graph-coloring paradigm, the compiler writer needs two additional mechanisms. First, the allocator needs an efficient technique for discovering k-colorings. Unfortunately, the problem of determining if a k-coloring exists for a particular graph is NP-Complete. Thus, register allocators use fast approximations that are not guaranteed to find a k-coloring. Second, the allocator needs a strategy for handling the case when no color remains for a specific live range. Most coloring allocators approach this by rewriting the code to change the allocation problem. The allocator picks one or more live ranges to modify. It either spills or splits the chosen live range. Spilling turns the chosen live range into a set of tiny live ranges, one at each definition and use of the original live range. Splitting breaks the live range into smaller, but non-trivial pieces. In either case, the transformed code performs the same computation, but has a different interference graph. If the changes are effective, the new interference graph is easier to color.

10.4.4 Top-down Coloring

A top-down, graph-coloring, global register allocator uses low-level information to assign colors to individual live ranges, and high-level information to select the order in which it colors live ranges. To find a color for a specific live range, LR_i, the allocator tallies the colors already assigned to LR_i's neighbors in I. If the set of neighbors' colors is incomplete—that is, one or more colors are not used—the allocator can assign an unused color to LR_i. If the set of neighbors' colors is complete, then no color is available for LR_i and the allocator must use its strategy for uncolored live ranges.

To order the live ranges, the top-down allocator uses an external ranking. The priority-based, graph-coloring allocators rank live ranges by the estimated run-time savings that accrue from keeping the live range in a register. These estimates are analogous to the spill-costs described in Section 10.4.2. The top-down global allocator uses registers for the most important values, as identified by these rankings.

The allocator considers the live ranges, in rank order, and attempts to assign a color to each of them. If no color is available for a live range, the allocator invokes the spilling or splitting mechanism to handle the uncolored live range. To improve the process, the allocator can partition the live ranges into two sets—constrained live ranges and unconstrained live ranges. A live range is constrained if it has k or more neighbors—that is, it has degree $\geq k$ in I. (We denote "degree of LR_i" as LR_i°, so LR_i is constrained if and only if $\text{LR}_i^\circ \geq k$.) Constrained live ranges are colored first, in rank order. After all constrained live ranges have been handled, the unconstrained live ranges are colored, in any order. An unconstrained live range must receive a color. When $\text{LR}_i^\circ < k$, no assignment of colors to LR_i's neighbors can prevent LR_i from receiving a color.

By handling constrained live ranges first, the allocator avoids some potential spills. The alternative, working in a straight priority order, would let the allocator assign all available colors to unconstrained, but higher priority, neighbors of LR_i. This could force LR_i to remain uncolored, even though colorings of its unconstrained neighbors that leave a color for LR_i must exist.

Handling Spills When the top-down allocator encounters a live range that cannot be colored, it must either spill or split some set of live ranges to change the problem. Since all previously colored live ranges were ranked higher than the uncolored live range, it makes sense to spill the uncolored live range rather than a previously colored live range. The allocator can consider re-coloring one of the previously colored live ranges, but it must exercise care to avoid the full generality and cost of backtracking.

To spill LR_i, the allocator inserts a store after every definition of LR_i and a load before each use of LR_i. If the memory operations need registers, the allocator can reserve enough registers to handle them. The number of registers needed for this purpose is a function of the target machine's instruction set architecture. Reserving these registers simplifies spilling.

An alternative to reserving registers for spill code is to look for free colors at each definition and each use; this strategy can lead to a situation where the allocator must retroactively spill a previously colored live range. (The allocator would recompute interferences at each spill site and compute the set of neighbor's colors for the spill site. If this process does not discover an open color at each spill site (or reference to the live range being spilled), the allocator would spill the lowest priority neighbor of the spill site. The potential for recursively spilling already colored live ranges has led most implementors of top-down, priority-based allocators to reserve spill registers, instead.) The paradox, of course, is that reserving registers for spilling may cause spilling; not reserving those registers can force the allocator to iterate the entire allocation procedure.

Splitting the Live Range Spilling changes the coloring problem. The entire uncolored live range is broken into a series of tiny live ranges—so small that spilling them is counterproductive. A related way to change the problem is to take the uncolored live range and break it into pieces that are larger than a single reference. If these new live ranges interfere, individually, with fewer live ranges than the original live range, then the allocator may find colors for them. For example, if the new live ranges are unconstrained, colors must exist for them. This process, called *live-range splitting*, can lead to allocations that insert fewer loads and stores than would be needed to spill the entire live range.

The first top-down, priority-based coloring allocator broke the uncolored live range into single-block live ranges, counted interferences for each resulting live range, and then recombined live ranges from adjacent blocks when the combined live range remained unconstrained. It placed an arbitrary upper limit on the number of blocks that a split live range could span. Loads and stores were added at the starting and ending points of each split live range. The allocator spilled any split live ranges that remained uncolorable.

10.4.5 Bottom-up Coloring

Bottom-up, graph-coloring register allocators use many of the same mechanisms as the top-down global allocators. These allocators discover live ranges, build an interference graph, attempt to color it, and generate spill code when needed. The major distinction between top-down and bottom-up allocators lies in the mechanism used to order live ranges for coloring. Where the top-down allocator uses high-level information to select an order for coloring, the bottom-up allocators compute an order from detailed structural knowledge about the interference graph, I. These allocators construct a linear ordering in which to consider the

```
initialize stack
while ($N \neq \emptyset$)
    if $\exists\ n \in N$ with $n^\circ < k$
        node $\leftarrow n$
    else
        node $\leftarrow n$ picked from $N$
    remove node and its edges from $I$
    push node onto stack
```

Figure 10.8: Computing a bottom-up ordering

live ranges, and then assign colors in that order.

To order the live ranges, bottom-up, graph-coloring allocators rely on a familiar observation:

> *A live range with fewer than k neighbors must receive a color, independent of the assignment of colors to its neighbors.*

The top-down allocators use this fact to partition the live ranges into constrained and unconstrained nodes. The bottom-up allocators use it to compute the order in which live ranges will be assigned colors, using the simple algorithm shown in Figure 10.8. The allocator repeatedly removes a node from the graph and places it on the stack. It uses two distinct mechanisms to select the node to remove next. The first clause takes a node that is unconstrained in the graph from which it is removed. The second clause, invoked only when every remaining node is constrained, picks a node using some external criteria. When the loop halts, the graph is empty and the stack contains all the nodes in order of removal.

To color the graph, the allocator rebuilds the interference graph in the order represented by the stack—the reverse of the order in which the allocator removed them from the graph. It repeatedly pops a node n from the stack, inserts n and its edges back into I, and looks for a color that works for n. At a high-level, the algorithm looks like:

```
while (stack $\neq \emptyset$)
    node $\leftarrow$ pop(stack)
    insert node and its edges into $I$
    color node
```

To color a node n, the allocator tallies the colors of n's neighbors in the current approximation to I and assigns n an unused color. If no color remains for n, it is left uncolored.

When the stack is empty, I has been rebuilt. If every node received a color, the allocator declares success and rewrites the code, replacing live range names with physical registers. If any node remains uncolored, the allocator either spills the corresponding live range or splits it into smaller pieces. At this point, the classic bottom-up allocators rewrite the code to reflect the spills and splits, and repeat the entire process—finding live ranges, building I, and coloring it. The process repeats until every node in I receives a color. Typically, the allocator halts in a couple of iterations. Of course, a bottom-up allocator could reserve registers for spilling, as described with the top-down allocator. This would allow it to halt after a single pass.

Why does this work? The bottom-up allocator inserts each node back into the graph from which it was removed. Thus, if the node representing LR_i was removed from I because it was unconstrained at the time, it is re-inserted into an approximation to I where it is also unconstrained—and a color must exist for it. The only nodes that can be uncolored, then, are nodes removed from I using the spill metric in the second clause of Figure 10.8. These nodes are inserted into graphs where they have k or more neighbors. A color may exist for them. Assume that $n^\circ > k$ when the allocator inserts it into I. Those neighbors cannot all have distinct colors. They can have at most k colors. If they have precisely k colors, then the allocator finds no color for n. If, instead, they use one color, or $k-1$ colors, or any number between 1 and $k-1$, then the allocator discovers a color for n.

The removal process determines the order in which nodes are colored. This order is crucial, in that it determines whether or not colors are available. For nodes removed from the graph by virtue of having low degree in the current graph, the order is unimportant with respect to the remaining nodes. The order may be important with respect to nodes already on the stack; after all, the current node may have been constrained until some of the earlier nodes were removed. For nodes removed from the graph by the second criterion ("*node picked from N*"), the order is crucial. This second criterion is invoked only when every remaining node has k or more neighbors. Thus, the remaining nodes form a heavily connected subset of I. The heuristic used to "pick" the node is often called the *spill metric*. The original bottom-up, graph-coloring allocator used a simple spill metric. It picked the node that minimized the fraction

$$\frac{\textit{estimated cost}}{\textit{current degree.}}$$

This picks a node that is relatively inexpensive to spill but lowers the degree of many other nodes. (Each remaining node has more than $k-1$ neighbors.) Other spill metrics have been tried, including minimizing estimated cost, minimizing the number of inserted operations, and maximizing removed edges. Since the actual coloring process is fast relative to building I, the allocator might try several colorings, each using a different spill metric, and retain the best result.

10.4.6 Coalescing Live Ranges to Reduce Degree

A powerful coalescing phase can be built that uses the interference graph to determine when two live ranges that are connected by a copy can be coalesced, or combined. Consider the operation `i2i` $\text{LR}_i \Rightarrow \text{LR}_j$. If LR_i and LR_j do not otherwise interfere, the operation can be eliminated and all references to LR_j rewritten to use LR_i. This has several beneficial effects. It directly eliminates the

$$
\begin{array}{llll@{\qquad\qquad}llll}
\texttt{add} & \texttt{LR}_t\texttt{,LR}_u & \Rightarrow & \texttt{LR}_i & \texttt{add} & \texttt{LR}_t\texttt{,LR}_u & \Rightarrow & \texttt{LR}_{ij} \\
 & \texttt{...} & & & & \texttt{...} & & \\
\texttt{i2i} & \texttt{LR}_i & \Rightarrow & \texttt{LR}_j & & & & \\
\texttt{i2i} & \texttt{LR}_i & \Rightarrow & \texttt{LR}_k & \texttt{i2i} & \texttt{LR}_{ij} & \Rightarrow & \texttt{LR}_k \\
 & \texttt{...} & & & & \texttt{...} & & \\
\texttt{add} & \texttt{LR}_j\texttt{,LR}_w & \Rightarrow & \texttt{LR}_x & \texttt{add} & \texttt{LR}_{ij}\texttt{,LR}_w & \Rightarrow & \texttt{LR}_x \\
\texttt{add} & \texttt{LR}_k\texttt{,LR}_y & \Rightarrow & \texttt{LR}_z & \texttt{add} & \texttt{LR}_k\texttt{,LR}_y & \Rightarrow & \texttt{LR}_z \\
\\
 & \textit{Before} & & & & \textit{After} & &
\end{array}
$$

Figure 10.9: Coalescing live ranges

copy operation, making the code smaller and, potentially, faster. It reduces the degree of any LR_k that interfered with both LR_i and LR_j. It shrinks the set of live ranges, making I and many of the data structures related to I smaller. Because these effects help in allocation, coalescing is often done before the coloring stage in a global allocator.

Notice that forming LR_{ij} cannot increase the degree of any of its neighbors in I. It can decrease their degree, or leave it unchanged, but it cannot increase their degree.

Figure 10.9 shows an example. The relevant live ranges are LR_i, LR_j, and LR_k. In the original code, shown on the left, LR_j is live at the definition of LR_k, so they interfere. However, neither LR_j nor LR_k interfere with LR_i, so both of the copies are candidates for coalescing. The fragment on the right shows the result of coalescing LR_i and LR_j to produce LR_{ij}.

Because coalescing two live ranges can prevent subsequent coalescing with other live ranges, order matters. In principle, the compiler should coalesce the most heavily executed copies first. In practice, allocators coalesce copies in order by the loop nesting depth of the block where the copy is found. Coalescing works from deeply nested to least deeply nested, on the theory that this gives highest priority to eliminating copy operations in innermost loops.

To perform coalescing, the allocator walks each block and examines any copy instructions. If the source and destination live ranges do not interfere, it combines them, eliminates the copy, and updates I to reflect the combination. The allocator can conservatively update I to reflect the change by moving all edges from the node for the destination live range to the node representing the source live range. This update, while not precise, allows the allocator to continue coalescing. In practice, allocators coalesce every live range allowed by I, then rewrite the code, rebuild I, and try again. The process typically halts after a couple of rounds of coalescing. It can produce significant reductions in the size of I. Briggs shows examples where coalescing eliminates up to one third of the live ranges.

Figure 10.9 also shows the conservative nature of the interference graph update. Coalescing LR_i and LR_j into LR_{ij} actually eliminates an interference.

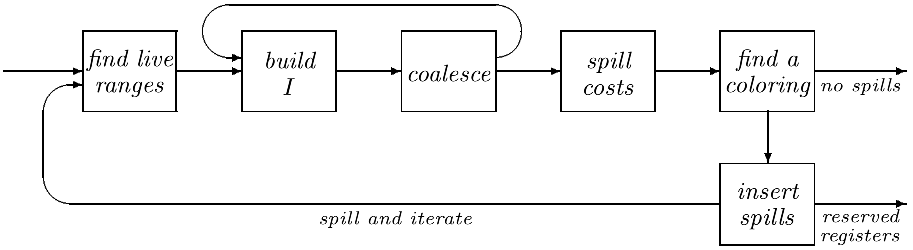


Figure 10.10: Structure of the coloring allocators

Careful scrutiny of the *after* fragment reveals that LR_{ij} does not interfere with LR_k, since LR_k is not live at the definition of LR_{ij} and the copy defining LR_k introduces no interference between its source and its destination. Thus, rebuilding I from the transformed code reveals that, in fact, LR_{ij} and LR_k can be coalesced. The conservative update left intact the interference between LR_j and LR_k, so it unnecessarily prevented the allocator from coalescing LR_{ij} and LR_k.

10.4.7 Review and Comparison

Both the top-down and the bottom-up coloring allocator work inside the same basic framework, shown in Figure 10.10. They find live ranges, build the interference graph, coalesce live ranges, compute spill costs on the coalesced version of the code, and attempt a coloring. The build-coalesce process is repeated until it finds no more opportunities. After coloring, one of two situations occurs. If every live range receives a color, then the code is rewritten using physical register names and allocation terminates. If some live ranges remain uncolored, then spill code is inserted.

If the allocator has reserved registers for spilling, then the allocator uses those registers in the spill code, rewrites the colored registers with their physical register names, and the process terminates. Otherwise, the allocator invents new virtual register names to use in spilling and inserts the necessary loads and stores to accomplish the spill. This changes the coloring problem slightly, so the entire allocation process is repeated on the transformed code. When all live ranges have a color, the allocator maps colors onto registers and rewrites the code into its final form.

Of course, a top-down allocator could adopt the spill-and-iterate philosophy used in the bottom-up allocator. This would eliminate the need to reserve registers for spilling. Similarly, a bottom-up allocator could reserve several registers for spilling and eliminate the need for iterating over the entire allocation process. Spill-and-iterate trades additional compile time for a tighter allocation, presumably using less spill code. Reserving registers produces a looser allocation with improved speed.

The top-down allocator uses its priority ranking to order all the constrained nodes. It colors the unconstrained nodes in arbitrary order, since the order cannot change the fact that they receive a color. The bottom-up allocator constructs an order in which most nodes are colored in a graph where they are unconstrained. Every node that the top-down allocator classifies as unconstrained is colored by the bottom-up allocator, since it is unconstrained in the original version of I and in each graph derived by removing nodes and edges from I. The bottom-up allocator, using its incremental mechanism for removing nodes and edges, classifies as unconstrained some of the nodes that the top-down allocator treats as constrained. These nodes may also be colored in the top-down allocator; there is no clear way of comparing their performance on these nodes without coding up both algorithms and running them.

The truly hard-to-color nodes are those that the bottom-up allocator removes from the graph with its spill metric. The spill metric is only invoked when every remaining node is constrained. These nodes form a densely connected subset of I. In the top-down allocator, these nodes will be colored in an order determined by their rank or priority. In the bottom-up allocator, the spill metric uses that same ranking, moderated by a measurement of how many other nodes have their degree lowered by each choice. Thus, the top-down allocator chooses to spill low priority, constrained nodes, while the bottom-up allocator spills nodes that are still constrained after all unconstrained nodes have been removed. From this latter set, it picks the node that minimizes the spill metric.

10.4.8 Other Improvements to Graph-coloring Allocation

Many variations on these two basic styles of graph-coloring register allocation have appeared in the literature. The first two address the compile-time speed of global allocation. The latter two address the quality of the resulting code.

Imprecise Interference Graphs The original top-down, priority-based allocator used an imprecise notion of interference: live ranges LR_i and LR_j interfere if both are live in the same basic block. This necessitated a prepass that performed local allocation to handle values that are not live across a block boundary. The advantage of this scheme is that building the interference graph is faster. The weakness of the scheme lies in its imprecision. It overestimates the degree of some nodes. It also rules out using the interference graph as a basis for coalescing, since, by definition, two live ranges connected by a copy interfere. (They must be live in the same block if they are connected by a copy operation.)

Breaking the Graph into Smaller Pieces If the interference graph can be separated into components that are not connected, those disjoint components can be colored independently. Since the size of the bit-matrix is $\mathbf{O}(N^2)$, breaking it into independent components saves both space and time. One way to split the graph is to consider non-overlapping register classes separately, as with floating-point registers and integer registers. A more complex alternative for large codes is to discover clique separators that divide the interference graph into several disjoint pieces. For large enough graphs, using a hash-table instead of the bit-matrix may improve both speed and space, although the choice of a hash-function has a critical impact on both.

Conservative Coalescing When the allocator coalesces two live ranges, LR_i and LR_j, the new live range, LR_{ij}, can be more constrained than either LR_i or LR_j. If LR_i and LR_j have distinct neighbors, then $\text{LR}^\circ_{ij} > max(\text{LR}^\circ_i, \text{LR}^\circ_j)$. If $\text{LR}^\circ_{ij} < k$, then creating LR_{ij} is strictly beneficial. However, if $\text{LR}^\circ_i < k$ and $\text{LR}^\circ_j < k$, but $\text{LR}^\circ_{ij} \geq k$, then coalescing LR_i and LR_j can make I harder to color without spilling. To avoid this problem, some compiler writers have used a limited form of coalescing called *conservative coalescing.* In this scheme, the allocator only combines LR_i and LR_j when $\text{LR}^\circ_{ij} < k$. This ensures that coalescing LR_i and LR_j does not make the interference graph harder to color.

If the allocator uses conservative coalescing, another improvement is possible. When the allocator reaches a point where every remaining live range is constrained, the basic algorithm selects a spill candidate. An alternative approach is to reapply coalescing at this point. Live ranges that were not coalesced because of the degree of the resulting live range may well coalesce in the reduced graph. Coalescing may reduce the degree of nodes that interfere with both the source and destination of the copy. Thus, this *iterated coalescing* can remove additional copies and reduce the degree of nodes. It may create one or more unconstrained nodes, and allow coloring to proceed. If it does not create any unconstrained nodes, spilling proceeds as before.

Spilling Partial Live Ranges As described, both global allocators spill entire live ranges. This 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 can find the regions where spilling a live range is productive—that is, the spill frees a register in a region where the register is needed. The splitting scheme described for the top-down allocator achieved this result by considering each block in the spilled live range separately. In a bottom-up allocator, similar results can be achieved by spilling only in the region of interference. One technique, called *interference region spilling*, identifies a set of live ranges that interfere in the region of high demand and limits spilling to that region [9]. The allocator can estimate the cost of several spilling strategies for the interference region and compare those costs against the standard, spill-everywhere approach. By letting the alternatives compete on an estimated cost basis, the allocator can improve overall allocation.

Live Range Splitting Breaking a live range into pieces can improve the results of coloring-based register allocation. In principle, splitting harnesses two distinct effects. If the split live ranges have lower degree than the original, they may be easier to color—possibly, unconstrained. If some of the split live ranges have high degree and, therefore, spill, then splitting may prevent spilling other pieces with lower degree. As a final, pragmatic effect, splitting introduces spills at the points where the live range is broken. Careful selection of those split points can control the placement of some spill code—for example, outside loops rather than inside loops.

Many approaches to splitting have been tried. Section 10.4.4 described an approach that breaks the live range into blocks and coalesces them back together when doing so does not change the allocator's ability to assign a color. Several approaches that use properties of the control-flow graph to choose split points have been tried. Results from many have been inconsistent [13], however two particular techniques show promise. A method called *zero-cost splitting* capitalizes on holes in the instruction schedule to split live ranges and improve both allocation and scheduling [39]. A technique called *passive splitting* uses a directed interference graph to determine where splits should occur and selects between splitting and spilling based on the estimated cost of each [26].

10.5 Regional Register Allocation

Even with global information, the global register allocators sometimes make decisions that result in poor code quality locally. To address these shortcomings, several techniques have appeared that are best described as regional allocators. They perform allocation and assignment over regions larger than a single block, but smaller than the entire program.

10.5.1 Hierarchical Register Allocation

The register allocator implemented in the compiler for the first Tera computer uses a hierarchical strategy. It analyzes the control-flow graph to construct a hierarchy of regions, or tiles, that capture the control flow of the program. Tiles include loops and conditional constructs; the tiles form a tree that directly encodes nesting and hierarchy among the tiles.

The allocator handles each tile independently, starting with the leaves of the tile tree. Once a tile has been allocated, summary information about the allocation is available for use in performing allocation for its parent in the tile tree—the enclosing region of code. This hierarchical decomposition makes allocation and spilling decisions more local in nature. For example, a particular live range can reside in different registers in distinct tiles. The allocator inserts code to reconcile allocations and assignments at tile boundaries. A preferencing mechanism attempts to reduce the amount of reconciliation code that the allocator inserts.

This approach has two other benefits. First, the algorithm can accommodate customized allocators for different tiles. For example, a version of the bottom-up local allocator might be used for long blocks, and an allocator designed for software pipelining applied to loops. The default allocator is a bottom-up coloring allocator. The hierarchical approach irons out any rough edges between these diverse allocators. Second, the algorithm can take advantage of parallelism on the machine running the compiler. Unrelated tiles can be allocated in parallel; serial dependences arise from the parent-child relationship in the tile tree. On the Tera computer, this reduced the running time of the compiler.

The primary drawback of the hierarchical approach lies in the code generated at the boundaries between tiles. The preferencing mechanisms must eliminate copies and spills at those locations, or else the tiles must be chosen so that those locations execute less frequently than the code inside the tiles. The quality of allocation with a hierarchical scheme depends heavily on what happens in these transitional regions.

10.5.2 Probabilistic Register Allocation

The probabilistic approach tries to address the shortcomings of graph-coloring global allocators by trying to generalize the principles that underlying the bottom-up, local allocator. The bottom-up, local allocator selects values to spill based on the distance to their next use. This distance can be viewed as an approximation to the probability that the value will remain in a register until its next use; the value with the largest distance has the lowest probability of staying in a register. From this perspective, the bottom-up, local allocator always spills the value least likely to keep its register. (*making it a self-fulfilling prophecy?*) The probabilistic technique uses this strong local technique to perform an initial allocation. It then uses a combination of probabilities and estimated benefits to make inter-block allocation decisions.

The global allocation phase of the probabilistic allocator breaks the code into regions that reflect its loop structure. It estimates both the expected benefit from keeping a live range in a register and the probability that the live range will stay in a register throughout the region. The allocator computes a merit rank for each live range as the product of its expected benefit and its global probability. It allocates a register for the highest ranking live range, adjusts probabilities for other live ranges to reflect this decision, and repeats the process. (Allocating LR_i to a register decreases the probability that conflicting live ranges can receive registers.) Allocation proceeds from inner loops to outer loops, and iterates until no uses remain in the region with probability greater than zero.

Once all allocation decisions have been made, it uses graph coloring to perform assignment. It makes a clear separation of allocation from assignment; allocation is performed using the bottom-up local method, followed by a probabilistic, inter-block analog for global allocation decisions.

10.5.3 Register Allocation via Fusion

The fusion-based register allocator presents another model for using regional information to drive global register allocation. The fusion allocator partitions the code into regions, constructs interference graphs for each region, and then fuses regions together to form the global interference graph. A region can be a single block, an arbitrary set of connected blocks, a loop, or an entire function. Regions are connected by control-flow edges.

The first step in the fusion allocator, after region formation, ensures that the interference graph for each region is k-colorable. To accomplish this, the allocator may spill some values inside the region. The second step merges the disjoint interference graphs for the code to form a single global interference graph. This is the critical step in the fusion allocator, because the fusion operator maintains k-colorability. When two regions are fused, the allocator may need to split some live ranges to maintain k-colorability. It relies on the observation that only values live along an edge joining the two regions affect the new region's colorability. To order the fusion operations, it relies on a priority ordering of the edges determined when regions are formed. The final step of the fusion allocator assigns a physical register to each allocated live range. Because the graph-merging operator maintains colorability, the allocator can use any of the standard graph-coloring techniques.

The strength of the fusion allocator lies in the fusion operator. By only splitting live ranges that are live across the edge or edges being combined, it localizes the inter-region allocation decisions. Because fusion is applied in edge-priority order, the code introduced by splitting tends to occur on low-priority edges. To the extent that priorities reflect execution frequency, this should lead to executing fewer allocator-inserted instructions.

Clearly, the critical problem for a fusion-based allocator is region formation. If the regions reflect execution frequencies—that is, group together heavily executed blocks and separate out blocks that execute infrequently, then it can force spilling into those lower frequency blocks. If the regions are connected by edges across which few values are live, the set of instructions introduced for splitting can be kept small. However, if the chosen regions fail to capture some of these properties, the rationalization for the fusion-based approach breaks down.

10.6 Harder Problems

This chapter has presented a selection of algorithms that attack problems in register allocation. It has not, however, closed the book on allocation. Many harder problems remain; there is room for improvement on several fronts.

10.6.1 Whole-program Allocation

The algorithms presented in this chapter all consider register allocation within the context of a single procedure. Of course, whole programs are built from multiple procedures. If global allocation produces better results than local allocation by considering a larger scope, then should the compiler writer consider performing allocation across entire programs? Just as the problem changes significantly when allocation moves from a single block to an entire procedure, it changes in important ways when allocation moves from single procedures to entire programs. Any whole-program allocation scheme must deal with the following issues.

To perform whole-program allocation, the allocator must have access to the code for the entire program. In practice, this means performing whole-program allocation at link-time. (While whole-program analyses and transformations can be applied before link-time, many issues that complicate an implementation of such techniques can be effectively side-stepped by performing them at link-time.)

To perform whole-program allocation, the compiler must have accurate estimates of the relative execution frequencies of all parts of the program. Within a procedure, static estimates (such as, "a loop executes 10 times") have proven to be reasonable approximations to actual behavior. Across the entire program, this may no longer be true.

In moving from a single procedure to a scope that includes multiple proce-

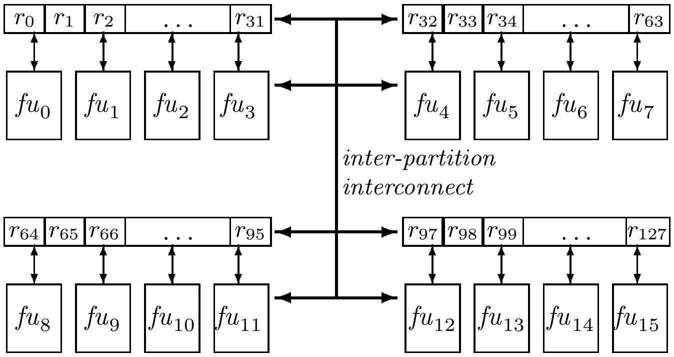


Figure 10.11: A clustered register-set machine

dures, the allocator must deal with parameter binding mechanisms and with the side effects of linkage conventions. Each of these creates new situations for the allocator. Call-by-reference parameter binding can link otherwise independent live ranges in distinct procedures together into a single interprocedural live range. Furthermore, they can introduce ambiguity into the memory model, by creating multiple names that can access a single memory location. (See the discussion of "aliasing" in Chapter 13.) Register save/restore conventions introduce mandatory spills that might, for a single intraprocedural live range, involve multiple memory locations.

10.6.2 Partitioned Register Sets

New complications can arise from new hardware features. For example, consider the non-uniform costs that arise on machines with partitioned register sets. As the number of functional units rises, the number of registers required to hold operands and results rises. Limitations arise in the hardware logic required to move values between registers and functional units. To keep the hardware costs manageable, some architects have partitioned the register set into smaller register files and clustered functional units around these partitions. To retain generality, the processor typically provides some limited mechanism for moving values between clusters. Figure 10.11 shows a highly abstracted view of such a processor. Assume, without loss of generality, that each cluster has an identical set of functional units and registers, except for their unique names.

Machines with clustered register sets layer a new set of complications onto the register assignment problem. In deciding where to place LR_i in the register set, the allocator must understand the availability of registers in each cluster, the cost and local availability of the inter-cluster transfer mechanism, and the specific functional units that will execute the operations that reference LR_i. For example, the processor might allow each cluster to generate one off-cluster register reference per cycle, with a limit of one off-cluster transfer out of each cluster each cycle. With this constraint, the allocator must pay attention to the placement of operations in clusters and in time. (Clearly, this requires attention from both the scheduler and the register allocator.) Another processor might require the code to execute a register-to-register move instruction, with limitations on the number of values moving in and out of each cluster in a single cycle. Under this constraint, cross-cluster use of a value requires extra instructions; if done on the critical path, this can lengthen overall execution time.

10.6.3 Ambiguous Values

A final set of complications to register allocation arise from shortcomings in the compiler's knowledge about the runtime behavior of the program. Many source-language constructs create ambiguous references (see Section 8.2), including array-element references, pointer-based references, and some call-by-reference parameters. When the compiler cannot determine that a reference is unambiguous, it cannot keep the value in a register. If the value is heavily used, this shortcoming in the compiler's analytical abilities can lead directly to poor run-time performance.

For some codes, heavy use of ambiguous values is a serious performance issue. When this occurs, the compiler may find it profitable to perform more detailed and precise analysis to remove ambiguity. The compiler might perform interprocedural data-flow analysis that reasons about the set of values that might be reachable from a specific pointer. It might rely on careful analysis in the front-end to recognize unambiguous cases and encode them appropriately in the IL. It might rely on transformations that rewrite the code to simplify analysis.

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 on pointer values to determine when a pointer-based value can be kept safely in a register throughout a loop nest, and to rewrite the code so that the value is kept in a newly introduced temporary variable. Both of these transformations move functionality from the register allocator into earlier phases of the compiler. Because they increase the demand for registers, they increase the cost of a misstep during allocation, and, conversely, increase the need for stable, predictable allocation. Ideally, these techniques should be integrated into the allocator, to ensure a fair competition between these "promoted" values and other values that are candidates for receiving a register.

10.7 Summary and Perspective

Because register allocation is an important component of a modern compiler, it has received much attention in the literature. Strong techniques exist for local allocation, for global allocation, and for regional allocation. Because the underlying problems are almost all NP-Complete, the solutions tend to be sensitive to small decisions, such as how ties between identically ranked choices are broken.

We have made progress on register allocation by resorting to paradigms that give us leverage. 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 problem. In fact, most of the improvements to the coloring allocators have come from attacking the points where the coloring paradigm does not accurately reflect the underlying problem, such as live range splitting, better cost models, and improved methods for live range splitting.

Questions

1. The top-down local allocator is somewhat naive in its handling of values. It allocates one value to a register for the entire basic block.

   (a) An improved version might calculate live ranges within the block and allocate values to registers for their live ranges. What modifications would be necessary to accomplish this?

   (b) A further improvement might be to split the live range when it cannot be accommodated in a single register. Sketch the data structures and algorithmic modifications that would be needed to (1) break a live range around an instruction (or range of instructions) where a register is not available, and to (2) re-prioritize the remaining pieces of the live range.

   (c) With these improvements, the frequency count technique should generate better allocations. How do you expect your results to compare with using Best's algorithm? Justify your answer.

2. When a graph-coloring global allocator reaches the point where no color is available for a particular live range, LR_i, it spills or splits that live range. As an alternative, it might attempt to re-color one or more of LR_i's neighbors. Consider the case where $\langle \text{LR}_i, \text{LR}_j \rangle \in I$ and $\langle \text{LR}_i, \text{LR}_k \rangle \in I$, but $\langle \text{LR}_j, \text{LR}_k \rangle \notin I$. If LR_j and LR_k have already been colored, and have received different colors, the allocator might be able to re-color one of them to the other's color, freeing up a color for LR_i.

   (a) Sketch an algorithm for discovering if a legal and productive re-coloring exists for LR_i.

   (b) What is the impact of your technique on the asymptotic complexity of the register allocator?

   (c) Should you consider recursively re-coloring LR_k's neighbors? Explain your rationale.

3. The description of the bottom-up global allocator suggests inserting spill code for *every* definition and use in the spilled live range. The top-down

global allocator first breaks the live range into block-sized pieces, then combines those pieces when the result is unconstrained, and finally, assigns them a color.

(a) 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 bottom-up global allocator that avoids this problem.

(b) If a given block has too many overlapping live ranges, then splitting a spilled live range does little to address the problem in that block. Suggest a mechanism (other than local allocation) to improve the behavior of the top-down global allocator inside blocks with high demand for registers.

Chapter Notes

Best's algorithm, detailed in Section 10.2.2 has been rediscovered repeatedly. Backus reports that Best described the algorithm to him in the mid-1950's, making it one of the earliest algorithms for the problem [3, 4]. Belady used the same ideas in his offline page replacement algorithm, MIN, and published it in the mid-1960's [8]. Harrison describes these ideas in connection with a regional register allocator in the mid-1970's [36]. Fraser and Hanson used these ideas in the `lcc` compiler in the mid-1980s. [28]. Liberatore *et al.* rediscovered and reworked this algorithm in the late 1990s [42]. They codified the notion of spilling clean values before spilling dirty values.

Frequency counts have a long history in the literature. . . .

The connection between graph coloring and storage allocation problems that arise in a compiler was originally suggested by the Soviet mathematician Lavrov [40]. He suggested building a graph that represented conflicts in storage assignment, enumerating its various colorings, and using the coloring that required the fewest colors. The Alpha compiler project used coloring to pack data into memory [29, 30].

Top-down graph-coloring begins with Chow. His implementation worked from a memory-to-memory model, so allocation was an optimization that improved the code by eliminating loads and stores. His allocator used an imprecise interference graph, so the compiler used another technique to eliminate extraneous copy instructions. The allocator pioneered live range splitting, using the scheme described on page 273 Larus built a top-down, priority-based allocator for SPUR-LISP. It used a precise interference graph and operated from a register-to-register model.

The first coloring allocator described in the literature was due to Chaitin and his colleagues at IBM [22, 20, 21]. The description of a bottom-up allocator in Section 10.4.5 follows Chaitin's plan, as modified by Briggs *et al.* [17]. Chaitin's contributions include the fundamental definition of interference, the algorithms for building the interference graph, for coalescing, and for handling spills. Briggs modified Chaitin's scheme by pushing constrained live ranges onto the stack rather than spilling them directly; this allowed Briggs' allocator to color a node with many neighbors that used few colors. Subsequent improvements in bottom-up coloring have included better spill metrics [10], methods for rematerializing simple values [16], iterated coalescing [33], methods for spilling partial live ranges [9], and methods for live range splitting [26]. The large size of precise interference graphs led Gupta, Soffa, and Steele to work on splitting the graph with clique separators [34]. Harvey *et al.* studied some of the tradeoffs that arise in building the interference graph [25].

The problem of modeling overlapping register classes, such as singleton registers and paired registers, requires the addition of some edges to the interference graph. Briggs *et al.* describe the modifications to the interference graph that are needed to handle several of the commonly occurring cases [15]. Cytron & Ferrante, in their paper "What's in a name?", give a polynomial-time algorithm for performing register assignment. It assumes that, at each instruction, $| \textsc{Live} | \; < \; k$. This corresponds to the assumption that the code has already been allocated.

Beatty published one of the early papers on regional register allocation [7]. His allocator performed local allocation and then used a separate technique to remove unneeded spill code at boundaries between local allocation regions—particularly at loop entries and exits. The hierarchical coloring approach was described by Koblenz and Callahan and implemented in the compiler for the Tera computer [19]. The probabilistic approach was presented by Proebsting and Fischer[44].

Chapter 11

Instruction Scheduling

11.1 Introduction

The order in which operations are presented for execution can have a significant affect on the length of time it takes to execute them. Different operations may take different lengths of time. The memory system may take more than one cycle to deliver operands to the register set or to one of the functional units. The functional units themselves may take several execution cycles to deliver the results of a computation. If an operation tries to reference a result before it is ready, the processor typically delays the operation's execution until the value is ready—that is, it *stalls*. The alternative, used in some processors, is to assume that the compiler can predict these stalls and reorder operations to avoid them. If no useful operations can be inserted to delay the operation, the compiler must insert one or more NOPs. In the former case, referencing the result too early causes a performance problem. In the latter case, the hardware assumes that this problem never happens, so when it does, the computation produces incorrect results. In either case, the compiler should carefully consider the ordering of instructions to avoid the problem.

Many processors can initiate execution on more than one operation in each cycle. The order in which the operations are presented for execution can determine the number of operations started, or issued, in a cycle. Consider, for example, a simple processor with one integer functional unit and one floating-point functional unit and a compiled loop that consists of one hundred integer operations and one hundred floating-point operations. If the compiler orders the operations so that the first seventy five operations are integer operations, the floating-point unit will sit idle until the processor can (finally) see some work for the floating-point unit. If all the operations were independent (an unrealistic assumption), the best order would be to alternate operations between the two units.

Most processors that issue multiple operations in each cycle have a simple algorithm to decide how many operations to issue. In our simple two functional unit machine, for example, the processor might examine two instructions at a

Start				
1	loadAI	r_0, 0	⇒	r_1
4	add	r_1, r_1	⇒	r_1
5	loadAI	r_0, 8	⇒	r_2
8	mult	r_1, r_2	⇒	r_1
9	loadAI	r_0, 16	⇒	r_2
12	mult	r_1, r_2	⇒	r_1
13	loadAI	r_0, 24	⇒	r_2
16	mult	r_1, r_2	⇒	r_1
18	storeAI	r_1	⇒	r_0,0

Original code

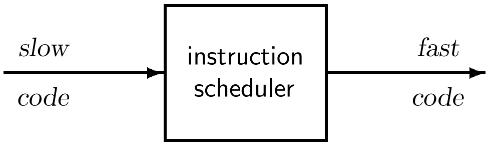
Start				
1	loadAI	r_0, 0	⇒	r_1
2	loadAI	r_0, 8	⇒	r_2
3	loadAI	r_0, 16	⇒	r_3
4	add	r_1, r_1	⇒	r_1
5	mult	r_1, r_2	⇒	r_1
6	loadAI	r_0, 24	⇒	r_2
7	mult	r_1, r_3	⇒	r_1
9	mult	r_1, r_2	⇒	r_1
11	storeAI	r_1	⇒	r_0,0

Scheduled code

Figure 11.1: Scheduling Example From Introduction

time. It would issue the first operation to the appropriate functional unit, and, if the second operation can execute on the other unit, issue it. If both operations need the same functional unit, the processor must delay the second operation until the next cycle. Under this scheme, the speed of the compiled code depends heavily on instruction order.

This kind of sensitivity to the order of operations suggests that the compiler should reorder operations in a way that produces faster code. This problem, reordering a sequence of operations to improve its execution time on a specific processor, has been called *instruction scheduling.* Conceptually, an instruction scheduler looks like



The primary goals of the instruction scheduler are to preserve the meaning of the code that it receives as input, to minimize execution time by avoiding wasted cycles spent in interlocks and stalls, and to avoid introducing extra register spills due to increased variable lifetimes. Of course, the scheduler should operate efficiently.

This chapter examines the problem of instruction scheduling, and the tools and techniques that compilers use to solve it. The next several subsections provide background information needed to discuss scheduling and understand both the algorithms and their impact.

11.2 The Instruction Scheduling Problem

Recall the example given for instruction scheduling in Section 1.3. Figure 11.1 reproduces it. The column labelled "*Start*" shows the cycle in which each operation executes. Assume that, the processor has a single functional unit; memory operations take three cycles; a `mult` takes two cycles; all other operations complete in a single cycle; and r_0 holds the activation record pointer (ARP). Under these parameters, the original code, shown on the left, takes twenty cycles.

The scheduled code, shown on the right, is much faster. It separates long-latency operations from operations that reference their results. This allows operations that do not depend on these results to execute concurrently. The code issues load operations in the first three cycles; the results are available in cycles 4, 5, and 6 respectively. This requires an extra register, r_3, to hold the result of the third concurrently executing load operation, but it allows the processor to perform useful work while waiting for the first arithmetic operand to arrive. In effect, this hides the latency of the memory operations. The same idea, applied throughout the block, hides the latency of the `mult` operation. The reordering reduces the running time to thirteen cycles, a thirty-five percent improvement.

Not all blocks are amenable to improvement in this fashion. Consider, for example, the following block that computes x^{256}:

Start		
1	`loadAI`	r_0,@x ⇒ r_1
2	`mult`	r_1,r_1 ⇒ r_1
4	`mult`	r_1,r_1 ⇒ r_1
6	`mult`	r_1,r_1 ⇒ r_1
8	`storeAI`	r_1 ⇒ r_0,@x

The three `mult` operations have long latencies. Unfortunately, each instruction uses the result of the previous instruction. Thus, the scheduler can do little to improve this code because it has no independent instructions that can be issued while the `mult`s are executing. Because it lacks independent operations that it can execute in parallel, we say that this block has no *instruction-level parallelism* (ILP). Given enough ILP, the scheduler can hide memory latency and functional-unit latency.

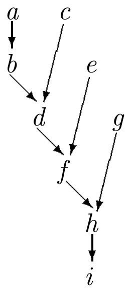
Informally, instruction scheduling is the process whereby a compiler reorders the operations in the compiled code in an attempt to decrease its running time. The instruction scheduler takes as input an ordered list of instructions; it produces as output a list of the same instructions.[1] The scheduler assumes a fixed set of operations—that is, it does not add operations in the way that the register allocator adds spill code. The scheduler assumes a fixed name space—it does not change the number of enregistered values, although a scheduler might perform some renaming of specific values to eliminate conflicts. The scheduler expresses its results by rewriting the code.

To define scheduling more formally requires introduction of a precedence graph $P = (N, E)$ for the code. Each node $n \in N$ is an instruction in the input code fragment. An edge $e = (n_1, n_2) \in E$ if and only if n_2 uses the result of n_1 as an argument. In addition to its edges, each node has two attributes, a *type*

[1] Throughout the text, we have been careful to distinguish between operations—individual commands to a single functional unit—and instructions—all of the operations that execute in a single cycle. Here, we mean "instructions."

```
a : loadAI   r0, 0   ⇒ r1
b : add      r1, r1  ⇒ r1
c : loadAI   r0, 8   ⇒ r2
d : mult     r1, r2  ⇒ r1
e : loadAI   r0, 16  ⇒ r2
f : mult     r1, r2  ⇒ r1
g : loadAI   r0, 24  ⇒ r2
h : mult     r1, r2  ⇒ r1
i : storeAI  r1      ⇒ r0,0
```

Example code



Its precedence graph

Figure 11.2: Precedence Graph for the Example

and a *delay*. For a node n, the instruction corresponding to n must execute on a functional unit of type *type*(n) and it requires *delay*(n) cycles to complete. The example in Figure 11.1 produces the precedence graph shown in Figure 11.2.

Nodes with no predecessors in the precedence graph, such as a, c, e, and g in the example, are called *leaves* of the graph. Since the leaves depend on no other operations, they can be scheduled at any time. Nodes with no successors in the precedence graph, such as i in the example, are called *roots* of the graph. A precedence graph can have multiple roots. The roots are, in some sense, the most constrained nodes in the graph because they cannot execute until all of their ancestors execute. With this terminology, it appears that we have drawn the precedence graph upside down—at least with relationship to the syntax trees and abstract syntax trees used earlier. Placing the leaves at the top of the figure, however, creates a rough correspondence between placement in the drawing and eventual placement in the scheduled code. A leaf is at the top of the tree because it can execute early in the schedule. A root is at the bottom of the tree because it must execute after each its ancestors.

Given a precedence graph for its input code fragment, a schedule S maps each node $n \in N$ into a non-negative integer that denotes the cycle in which it should be issued, assuming that the first operation issues in cycle one. This provides a clear and concise definition of an instruction: the i^{th} instruction is the set of operations $n \ni S(n) = i$. A schedule must meet three constraints.

1. $S(n) \geq 0$, for each $n \in N$. This constraint forbids operations from being issued before execution starts. Any 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 $(n_1, n_2) \in E, S(n_1) + \mathit{delay}(n_1) \leq S(n_2)$. This constraint enforces correctness. It requires that an operation cannot be issued until the operations that produce its arguments have completed. A schedule that violates this rule changes the flow of data in the code and is likely to produce in-

correct results.

3. Each instruction contains no more operations of type t than the target machine can issue. This constraint enforces feasibility, since a schedule that violates it contains instructions that the target machine cannot possibly issue.[2]

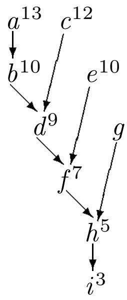
The compiler should only produce schedules that meet all three constraints.

Given a well-formed schedule that is both correct and feasible, the length of the schedule is simply the cycle number in which the last operation completes. This can be computed as

$$L(S) = \max_{n \in N}(S(n) + delay(n)).$$

Assuming that *delay* captures all of the operational latencies, schedule S should execute in $L(S)$ time.[3] With a notion of schedule length comes the notion of a *time-optimal* schedule. A schedule S_i is time-optimal if $L(S_i) \leq L(S_j)$, for all other schedules S_j.

The precedence graph captures important properties of the schedule. Computing the total delay along the paths through the graph exposes additional detail about the block. Annotating the precedence graph from Figure 11.1 with information about cumulative latency yields the following graph:



The path length from a node to the end of the computation is shown as a superscript on the node. The values clearly show that the path *abdfhi* is longest—it is the *critical path* that determines overall execution time.

How, then, should the compiler schedule this computation? An operation can only be scheduled when its operands are available for use. Thus, operations

[2] Some machines require that each instruction contain precisely the number of operations of type t that the hardware can issue. We think of these machines as VLIW computers. This scheme produces a fixed-length instruction with fixed fields and simplifies the implementation of instruction decode and dispatch in the processor. More flexible schemes allow instructions that leave some functional units idle or processing operations issued in earlier cycles.

[3] Of course, some operations have variable delays. For example, a load operation on a machine with a complex memory hierarchy may have an actual delay that ranges from zero cycles if the requested value is in cache to hundreds or thousands of cycles if the value is in distant memory. If the scheduler assumes the worst case delay, it risks idling the processor for large periods. If it assumes the best case delay, it will stall the processor on a cache miss.

a, *c*, *e*, and *g* are the initial candidates for scheduling. The fact that *a* lies on the critical path strongly suggests that operation it be scheduled first. Once *a* has been scheduled, the longest remaining path is *cdefhi*, suggesting that operation *c* be scheduled second. With a schedule of *ac*, *b* and *e* tie for the longest path. However, *b* needs the result of *a*, which will not be available until the fourth cycle. This makes *e* followed by *b*, the better choice. Continuing in this fashion leads to the schedule *acebdgfhi*. This matches the schedule shown on the left side of Figure 11.1.

However, the compiler cannot simply rearrange the instructions into the proposed order. Notice that operations *c* and *e* both define $\texttt{r}_2$. In reordering the code, the scheduler cannot move *e* before *d*, unless it renames the result of *e* to avoid the conflict with the definition of `r2` by *c*. This constraint arises not from the flow of data, as with the dependences modelled in the precedence graph. Instead, it arises from the need to avoid interfering with the dependences modelled in the graph. These constraints are often called *anti-dependences*.

The scheduler can produce correct code in two ways. It can discover the anti-dependences that exist in the input code and respect them in the final schedule, or it can rename values to avoid them. The example contains four anti-dependences: *e* to *c*, *e* to *d*, *g* to *e*, and *g* to *f*. All of them involve the redefinition of $\texttt{r}_2$. (Constraints exist based on $\texttt{r}_1$ as well, but any anti-dependence on $\texttt{r}_1$ duplicates a constraint that also arises from a dependence based on the flow of values.)

Respecting the anti-dependence changes the set of schedules that the compiler can produce. For example, it cannot move *e* before either *c* or *d*. This forces it to produce a schedule such as *acbdefghi*, which requires eighteen cycles. While this schedule is a ten percent improvement over the unscheduled code (*abcdefghi*), it is not competitive with the thirty-five percent improvement obtained by renaming to produce *acebdgfhi*, as shown on the right side of Figure 11.1.

The alternative is to systematically rename the values in the block to eliminate anti-dependences, to schedule, and then to perform register allocation and assignment. This approach frees the scheduler from the constraints imposed by anti-dependences; it creates the potential for problems if the scheduled code requires spill code. The act of renaming, however, does not change the number of live variables; it simply changes their names. It does, however, give the scheduler freedom to move definitions and uses in a way that increases the number of concurrently live values. If the scheduler drives this number too high, the allocator will need to insert spill code—adding long-latency operations and necessitating further scheduling.

The simplest renaming scheme assigns a new name to each value as it is produced. In the ongoing example, this produces the following code:

```
a : loadAI   r0, 0   ⇒ r1
b : add      r1, r1  ⇒ r2
c : loadAI   r0, 8   ⇒ r3
d : mult     r2, r3  ⇒ r4
e : loadAI   r0, 16  ⇒ r5
f : mult     r4, r5  ⇒ r1
g : loadAI   r0, 24  ⇒ r6
h : mult     r5, r6  ⇒ r7
i : storeAI  r7      ⇒ r0,0
```

This version of the code has the same pattern of definitions, and uses. However, the dependence relationships are expressed unambiguously in the code. It contains no anti-dependences, so naming constraints cannot arise.

Other Measures of Schedule Quality Schedules can be measured in terms other than time. Two schedules S_i and S_j for the same input code might produce different actual demand for registers—that is, the maximum number of live values in S_j may be less than in S_i. Both S_i and S_j might use the same space of register names, and the corresponding operations might use the same registers for operands and results. If the processor requires the scheduler to insert NOPs for idle functional units, then S_i might have fewer operations than S_j. This need not depend solely on schedule length. If the processor includes a variable-cycle NOP, as does the TMS320C6200, then bunching NOPs together produces fewer operations. Finally, S_j might require less power to execute on the target machine because it never uses one of the functional units, or because it causes fewer bit-transitions in the processor's instruction decoder.

What Makes Scheduling Hard? The scheduler must choose a cycle in which to issue each operation and it must determine, for each cycle in the execution, which operations can be issued. In balancing these two viewpoints, it must ensure that an operation only issues when its operands have been computed and are available. In a given cycle, more than one operations may meet this criterion, so the scheduler must choose between them. Since values are live from their definitions to their uses, the decisions it makes can shorten register lifetimes (if uses move closer to definitions) or lengthen them (if uses move away from their definitions). If too many values become live, the schedule may be infeasible because it requires too many registers. Balancing all of these issues, while searching for a time-optimal schedule, makes scheduling complex.

In practice, almost all 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 and more complex regions of code.

11.3 Local List Scheduling

List scheduling is a greedy, heuristic approach to scheduling the operations in a basic block. It has been the dominant paradigm for instruction scheduling since

Digression: How Does Scheduling Relate to Allocation?

The use of specific names can introduce anti-dependences that limit the scheduler's ability to reorder operations. The scheduler can avoid anti-dependences by renaming; however, renaming creates a need for the compiler to perform register assignment after scheduling. Instruction scheduling and register allocation interact in complex ways.

The core function of the instruction scheduler is to reorder operations. Since most operations use some values and produce a new value, changing the relative order of two instructions can change the lifetimes of some values. If operation a defines $\texttt{r}_1$ and b defines $\texttt{r}_2$, moving b before a can have several effects. It lengthens the lifetime of $\texttt{r}_2$ by one cycle, while shortening the lifetime of $\texttt{r}_1$. If one of a's operands is a last use, then moving b before a lengthens its lifetime. Symmetrically, if one of b's operands is a last use, the move shortens that operand's lifetime.

The net effect of moving b before a depends on the details of a and b, as well as the surrounding code. If none of the uses involved are last uses, then the swap has no net effect on demand for registers. (Each operation defines a register; swapping them changes the lifetimes of specific registers, but not the aggregate demand.)

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 memory operations when demand for registers is too high. Both these functions affect the ability of the scheduler to produce fast code. When the allocator maps a large virtual name space into the name space of target machine registers, it can introduce anti-dependences that constrain the scheduler. Similarly, when the allocator inserts spill code, it adds instructions to the code that must, themselves, be scheduled to hide their latencies.

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 we will treat them separately. In practice, most compilers treat them separately, as well.

the late 1970s, largely because it discovers reasonable schedules and adapts easily to changes in the underlying computer architectures. However, list scheduling describes an approach rather than a specific algorithm. Wide variation exists in how it is implemented and how it attempts to prioritize instructions for scheduling. This section explores the basic framework of list scheduling, as well as a couple of variations on the idea.

Classic list scheduling operates over a single basic block. Limiting our consideration to straight-line sequences of code allows us to ignore situations that can complicate scheduling. For example, with multiple blocks, an operand might depend on more than one previous definition; this creates uncertainty about when the operand is available for use. With several blocks in its scope, the scheduler might move an operation from one block to another. This cross-block code motion can force duplication of the operation—moving it into a successor block may create the need for a copy in each successor block. Similarly, it can cause an operation to execute on paths where it did not in the original code—moving it into a predecessor block puts it on the path to each of that block's successors. Restricting our consideration to the single block case avoids these complications.

To apply list scheduling to a block, the scheduler follows a four step plan:

1. *Rename to avoid anti-dependences.* To reduce the set of constraints on the scheduler, the compiler renames values. Each definition receives a unique name. This step is not strictly necessary. However, it lets the scheduler find some schedules that the anti-dependences would have prevented. It also simplifies the scheduler's implementation.

2. *Build a precedence graph, $\mathcal{P}$.* To build the precedence graph, the scheduler walks the block from bottom to top. For each operation, it constructs a node to represent the newly created value. It adds edges from that node to each node that uses the value. The edges are annotated with the latency of the current operation. (If the scheduler does not perform renaming, $\mathcal{P}$ must represent anti-dependence as well.)

3. *Assign priorities to each operation.* The scheduler will use these priorities to guide it as it picks from the set of available operations at each step. Many priority schemes have been used in list schedulers. The scheduler may compute several different scores for each node, using one as the primary ordering and the others to break ties between equally-ranked nodes. The most popular priority scheme uses the length of the longest latency-weighted path from the node to a root of $\mathcal{P}$. We will describe other priority schemes later.

4. *Iteratively select an operation and schedule it.* The central data structure of the list scheduling algorithm is a list of operations that can legally execute in the current cycle, called the ready list. Every operation on the ready list has the property that its operands are available. The algorithm starts at the first cycle in the block and picks as many operations to issue in that cycle as possible. It then advances the cycle counter and updates the ready list to reflect both the previously issued operations and the passage of time. It repeats this process until every operation has been scheduled.

Renaming and building $\mathcal{P}$ are straight forward. The priority computations typically involve a traversal of $\mathcal{P}$. The heart of the algorithm, and the key to understanding it, lies in the final step. Figure 11.3 shows the basic framework for this step, assuming that the target machine has a single functional unit.

The algorithm performs an abstract simulation of the code's execution. It ignores the details of values and operations to focus on the timing constraints

```
$Cycle \leftarrow 1$
$Ready \leftarrow$ leaves of $\mathcal{P}$
$Active \leftarrow \emptyset$

while ($Ready \cup Active \neq \emptyset$)
    if $Ready \neq \emptyset$ then
        remove an op from Ready
        $S(op) \leftarrow Cycle$
        $Active \leftarrow Active \cup op$

    $Cycle \leftarrow Cycle + 1$

    for each $op \in Active$
        if $S(op) + delay(op) \leq Cycle$ then
                  remove op from Active
            for each successor s of op in $\mathcal{P}$
                if s is ready then
                   $Ready \leftarrow Ready \cup s$
```

Figure 11.3: List Scheduling Algorithm

imposed by edges in $\mathcal{P}$. To accomplish this, it maintains a simulation clock, in the variable *Cycle*. *Cycle* is initialized to one, the first operation in the code, and incremented until every operation in the code has been assigned a time to execute.

The algorithm uses two lists to track operations. The first list, called the *Ready* list, holds any operations that can execute in the current cycle. If an operation is in *Ready*, all of its operands have been computed. Initially, *Ready* contains all the leaves of $\mathcal{P}$, since they depend on no other operations. The second list, called the *Active* list, holds any operations that were issued in an earlier cycle but have not yet completed. At each time step, the scheduler checks *Active* to find operations that have finished. As each operation finishes, the scheduler checks its successors s in $\mathcal{P}$ (the operations that use its result) to determine if all of the operands for s are now available. If they are, it adds s to *Ready*.

At each time step, the algorithm follows a simple discipline. It accounts for any operations completed in the previous cycle, then it schedules an operation for the current cycle. The details of the implementation slight obscure this structure because the first time step always has an empty *Active* list. The while loop begins in the middle of the first time-step. It picks an operation from *Ready* and schedules it. Next, it increments *Cycle* to begin the second time step. It updates *Active* and *Ready* for the beginning of the new cycle, then wraps around to the top of the loop, where it repeats the process of pick, increment, and update.

The process terminates when every operation has executed to completion. At the end, *Cycle* contains the simulated running time of the block. If all operations execute in the time specified by *delay*, and all operands to the leaves of $\mathcal{P}$ are available in the first cycle, this simulated running time should match the actual execution time.

An important question, at this point, is "how good is the schedule that this method generates?" The answer depends, in large part, on how the algorithm picks the operation to remove from the *Ready* list in each iteration. 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 $\mathcal{P}$, and our restriction ensures that there is at most 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 multiple operations are available at some point in the process.

In the ongoing example, $\mathcal{P}$ has four leaves: a, c, e, and g. With only one functional unit, the scheduler must select one of the four load operations to execute in the first cycle. This is the role of the priority computation—it assigns each node in $\mathcal{P}$ a set of one or more ranks that the scheduler uses to order the nodes for scheduling. The metric suggested earlier, the longest latency-weighted distance to a root in $\mathcal{P}$, corresponds to always choosing the node on the critical path for the current cycle in the current schedule. (Changing earlier choices might well change the current critical path.) To the limited extent that the impact of a scheduling priority is predictable, this scheme should provide balanced pursuit of the longest paths.

Efficiency Concerns To pick an operation from the *Ready* list, as described this far, requires a linear scan over *Ready*. This makes the cost of creating and maintaining *Ready* approach $\mathsf{O}(n^2)$. Replacing the list with a priority queue can reduce the cost of these manipulations to $\mathsf{O}(n\ log_2\ n)$, for a minor increase in the difficulty of implementation. If the

A similar approach can reduce the cost of manipulating the *Active* list. When the scheduler adds an operation to *Active*, it can assign it a priority equal to the cycle in which the operation completes. A priority queue that seeks the smallest priority will push all the operations completed in the current cycle to the front, for a small increase in cost over a simple list implementation.

Further improvement is possible with the implementation of *Active*. The scheduler can maintain a set of separate lists, one for each cycle in which an operation can finish. The number of lists required to cover all the operation latencies is $MaxLatency = max_{n \in \mathcal{P}}\, delay(n)$. When the compiler schedules operation n in *Cycle*, it adds n to *WorkList[(Cycle +* $delay(n)$*) mod* $MaxLatency$]. When it goes to update the *Ready* queue, all of the operations with successors to consider are found in *WorkList[Cycle mod* $MaxLatency$*]*. This schemes uses a small amount of extra space and some extra time to reduce the quadratic cost of searching *Active* to the linear cost of walking through and emptying the *WorkList*. (The number of operations in the *WorkList*s is identical to the number in the *Active*, so the space overhead is just the cost of having multiple *WorkList*s. The extra time comes from introducing the subscript calculations on *WorkList*, including the *mod* calculations.)

Caveats: As described, the local list scheduler assumes that all operands for leaves are available on entry to the block. This requires that previous blocks' schedules include enough slack time at the end for all operations to complete. With more contextual knowledge, the scheduler might prioritize leaves in a way that creates that slack time internally in the block.

The local list scheduler we have described assumes a single functional unit of a single type. Almost no modern computers fit that model. The critical loop of the iteration must select an operation for each functional unit, if possible. If it has several functional units of a given type, it may need to bias its choice for a specific operation by other features of the target machine—for example, if only one of the units can issue memory operations, loads and stores should be scheduled onto that unit first. Similarly, if the register set is partitioned (see Section 10.6.2), the scheduler may need to place an operation on the unit where its operands reside or in a cycle when the inter-partition transfer apparatus is free.

11.3.1 Other Priority Schemes

Instruction scheduling, even in the local case, is NP-complete for most realistic scenarios. As with many other NP-complete problems, greedy heuristic techniques like list scheduling produce fairly good approximations to the optimal solution. However, the behavior of these greedy techniques is rarely robust—small changes in the input may make large differences in the solution.

One algorithmic way of addressing this instability is careful *tie-breaking.* When two or more items have the same rank, the implementation chooses among them based on another priority ranking. (In contrast, the MAX function typically exhibits deterministic behavior—either it retains the first value of maximal rank or the last such value. This introduces a systematic bias toward earlier nodes or later nodes in the list.) A good list scheduler will use several different priority rankings to break ties. Among the many priority schemes that have been suggested in the literature are:

- A node's rank is the total length of the longest path that contains it. This favors, at each step, the critical path in the original code, ignoring intermediate decisions. This tends toward a depth-first traversal of $\mathcal{P}$.

- A node's rank is the number of immediate successors it has in $\mathcal{P}$. This encourages the scheduler to pursue many distinct paths through the graph—closer to a breadth first approach. It tends to keep more operations on the *Ready* queue.

- A node's rank is the total number of descendants it has in $\mathcal{P}$. This amplifies the effect seen in the previous ranking. Nodes that compute critical values for many other nodes are scheduled early.

- A node's rank is higher if it has long latency. This tends to schedule the long latency nodes early in the block, when more operations remain that might be used to cover the latency.

- A node's rank is higher if it contains the last use of a value. This tends to decrease demand for registers, by moving last uses closer to definitions.

Unfortunately, there is little agreement over either which rankings to use or which order to apply them.

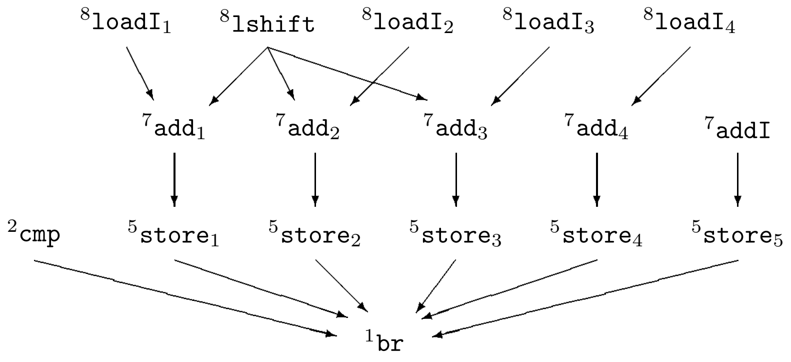
11.3.2 Forward versus Backward List Scheduling

An alternate formulation of list schedule works over the precedence graph in the opposite direction, scheduling from roots to leaves. The first operation scheduled executes in the last cycle of the block, and the last operation scheduled executes first. In this form, the algorithm is called *backward list scheduling*, making the original version *forward list scheduling*.

A standard practice in the compiler community states is to try several versions of list scheduling on each block and keep the shortest schedule. Typically, the compiler tries both forward and backward list scheduling; it may try more than one priority scheme in each direction. Like many tricks born of experience, this one encodes several important insights. First, list scheduling accounts for a small portion of the compiler's execution time. The compiler can afford to try several schemes if it produces better code. Notice that the scheduler can reuse most of the preparatory work—renaming, building $\mathcal{P}$, and some of the computed priorities. Thus, the cost of using multiple schemes amounts to repeating the iterative portion of the scheduler a small number of times. Second, the practice suggests that neither forward scheduling nor backward scheduling always wins. The difference between forward and backward list scheduling lies in the order in which operations are considered. If the schedule depends critically on the careful ordering of some small set of operations, the two directions may produce radically different results. If the critical operations occur near the leaves, forward scheduling seems more likely to consider them together, while backward scheduling must work its way through the remainder of the block to reach them. Symmetrically, if the critical operations occur near the roots, backward scheduling may examine them together while forward scheduling sees them in an order dictated by decisions made at the other end of the block.

To make this latter point more concrete, consider the example shown in Figure 11.4. It shows the precedence graph for a basic block found in the SPEC 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. (Violating this assumption could produce an incorrect value from a subsequent load operation.) Superscripts on nodes in the precedence graph give the latency from the node to the branch; subscripts differentiate between similar operations. The example assumes operation latencies that appear in the table below the precedence graph.

This example demonstrates the difference between forward and backward list scheduling. The five store operations take most of the time in the block. The



Opcode	`loadI`	`lshift`	`add`	`addI`	`cmp`	`store`
Latency	1	1	2	1	1	4

Figure 11.4: Precedence graph for a block from go

schedule that minimizes execution time must begin executing stores as early as possible.

Forward list scheduling, using latency-to-root for priority, executes the operations in priority order, except for the comparison. It schedules the five operations with rank eight, then the five operations with rank seven. It begins on the operations with weight 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 on the left side of Figure 11.5. Notice that the memory operations begin in cycle five, producing a schedule that issues the branch in cycle thirteen.

Using the same priorities with reverse list scheduling, the compiler first places the branch in the last slot of the block. The `cmp` precedes it by *delay*(`cmp`) $= 1$ cycle. The next operation scheduled is `store`$_1$ (by the left-to-right tie breaking rule). It is assigned the issue slot on the memory unit that is *delay*(`store`) $= 4$ cycles earlier. 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 `add`$_1$, two cycles before `store`$_1$. When the algorithm terminates, it has produces the schedule shown on the right side of Figure 11.5.

The schedule produced by the backward scheduler takes one fewer cycle than the schedule produced by the forward scheduler. It places the `addI` earlier in the block, allowing `store`$_5$ to issue in cycle four—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 achieves a different result. Using the same graph structure, one can construct a block where the forward scheduler beats the backward

Forward Schedule

	Int.	Int.	Mem.
1.	$\texttt{loadI}_1$	lshift	---
2.	$\texttt{loadI}_2$	$\texttt{loadI}_3$	---
3.	$\texttt{loadI}_4$	$\texttt{add}_1$	---
4.	$\texttt{add}_2$	$\texttt{add}_3$	---
5.	$\texttt{add}_4$	addI	$\texttt{store}_1$
6.	cmp	---	$\texttt{store}_2$
7.	---	---	$\texttt{store}_3$
8.	---	---	$\texttt{store}_4$
9.	---	---	$\texttt{store}_5$
10.	---	---	---
11.	---	---	---
12.	---	---	---
13.	br	---	---

Backward Schedule

	Int.	Int.	Mem.
1.	$\texttt{loadI}_4$	---	---
2.	addI	lshift	---
3.	$\texttt{add}_4$	$\texttt{loadI}_3$	---
4.	$\texttt{add}_3$	$\texttt{loadI}_2$	$\texttt{store}_5$
5.	$\texttt{add}_2$	$\texttt{loadI}_1$	$\texttt{store}_4$
6.	$\texttt{add}_1$	---	$\texttt{store}_3$
7.	---	---	$\texttt{store}_2$
8.	---	---	$\texttt{store}_1$
9.	---	---	---
10.	---	---	---
11.	cmp	---	---
12.	br	---	---

Figure 11.5: Schedules for the block from go

scheduler.

Why does this happen? The forward scheduler must place 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 handling it. By the time that the scheduler runs out of rank eight operations, other rank seven operations are available. In contrast, the backward scheduler places the addI before three of the rank eight operations—a result that the forward scheduler could not consider.

The list scheduling algorithm uses a greedy heuristic to construct a reasonable solution to the problem, but the ranking function does not encode complete knowledge of the problem. To construct a ranking that encoded complete knowledge, the compiler would need to consider all possible schedules—making the ranking process itself NP-Complete. Thus, an optimal ranking function is impractical, unless P = NP.

11.3.3 Why Use List Scheduling?

List scheduling has been the dominant algorithm for instruction scheduling for many years. The technique, in its forward and backward forms, uses a greedy heuristic approach to assign each operation an issue slot in the basic block. In practice, it produces excellent results for single blocks.

List scheduling is efficient. It removes an operation from the *Ready* queue once. It examines each operation for addition to the *Ready* queue once for each edge that enters its node in $\mathcal{P}$. If an operation has m operands, the scheduler visits its node m times. Each visit looks at each of its m operands, so the amount of work involved in placing it on the ready queue is $\mathsf{O}(m^2)$. However, for most operations, m is one or two, so this quadratic cost is trivial.

Digression: What About Out-of-Order Execution?

Some processors include hardware support for executing instructions out-of-order (OOO). We refer to such processors as *dynamically scheduled* machines. This feature is not new; for example, it appeared on the IBM 360/91. To support out-of-order execution, a dynamically-scheduled processor looks ahead in the instruction stream for operations that can execute before they would in a statically-scheduled processor. To do this, the dynamically-scheduled processor builds and maintains a portion of the precedence graph at run-time. It uses this piece of the precedence graph to discover when each operation can execute and issues each operation at the first legal opportunity.

When can an out-of-order processor improve on the static schedule? It cannot avoid waiting for a data dependence, since that represents the actual transmission of a value. It can issue an operation early precisely when the scheduler could have placed the operation earlier—that is, all of the operands are ready before the operation's appointed cycle. Taken over an entire block, this can snowball—one operation executing early may create opportunities for its descendants in $\mathcal{P}$. The result is a dynamic rewriting of the compiler-specified schedule.

Ooo execution does not eliminate the need for instruction scheduling. Because the lookahead window is finite, bad schedules can defy improvement. For example, a lookahead window of fifty operations will not let the processor execute a string of 100 integer operations followed by 100 floating-point operations in interleaved (integer, floating-point) pairs. It may, however, interleave shorter strings, say of length thirty. Ooo execution helps the compiler by improving good, but non-optimal, schedules.

A related processor feature is dynamic register renaming. This scheme provides the processor with more physical registers than the ISA allows the compiler to name. The processor can break anti-dependences that occur within its lookahead window by using physically distinct registers to implement two references connected by an anti-dependence.

List scheduling forms the basis for most algorithms that perform scheduling over regions larger than a single block. Thus, understanding its strengths and its weaknesses is important. Any improvement made to local list scheduling has the potential to improve the regional scheduling algorithms, as well.

11.4 Regional Scheduling

As we saw with register allocation, moving from single basic blocks to larger scopes can sometimes improve the quality of code that the compiler generates. With instruction scheduling, many different approaches have been proposed for regions larger than a block, but smaller than a whole procedure. This section examines three approaches that derive from list scheduling.

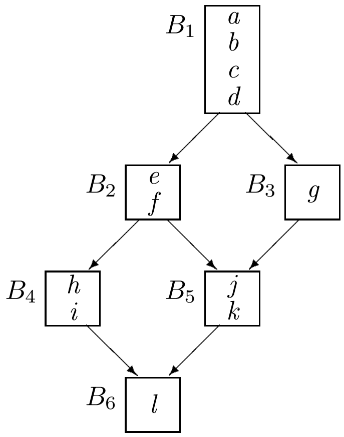


Figure 11.6: Extended basic block scheduling example

11.4.1 Scheduling Extended Basic Blocks

An extended basic block (EBB) consists of a series of blocks $b_1, b_2, b_3, \ldots b_n$ where b_1 has multiple predecessors in the control-flow graph and the other blocks b_i each have precisely one predecessor, $b_{i-1}, \forall\, 2 \leq i \leq n$. Extending list scheduling to handle an EBB allows the compiler to consider a longer run of execution during scheduling. Some care must be taken, however, to ensure correct behavior at interim exits from the EBB.

Consider the simple branching structure shown in Figure 11.6. It has several distinct EBBs: (B_1, B_2, B_4), (B_1, B_3), (B_5), and (B_6). Each of these EBBs could be subject to EBB scheduling. Notice, however, that the first two EBBs, (B_1, B_2, B_4), (B_1, B_3) share a common prefix—the block B_1. The compiler cannot schedule block B_1 in two conflicting ways.

The conflict over B_1 can exhibit itself in two ways. The scheduler might move an operation from B_1, for example c, past the branch at the end of B_1 and down into B_2. In this case, c would no longer be on the path from the entry of B_1 to the entry of B_3, so the scheduler would need to insert a copy of c into B_3 to maintain correctness along the path through B_3. Alternatively, the scheduler might move an operation from B_2, for example f, up into B_1. This puts f on the path from the entry of B_1 to the entry of B_3, where it did not previously execute. This lengthens the path through B_3, with the possibility of slowing down execution along that path.

(Without renaming to avoid anti-dependences, it might also change the correctness of that path. To perform renaming in the presence of control flow may require the insertion of copy operations along some of the edges in the control-flow graph in order to maintain correctness. See the discussion of translating from SSA-form back into a linear representation in Section 13.2.)

The compiler has several alternatives for managing these intra-block conflicts. To handle the problems that arise from downward motion, it can insert *compensation code* into blocks that are along exits from the EBB. Any time that it moves an operation from block B_i into one of B_i's successors, it must consider creating a copy of the instruction in every other successor of B_i. If the value defined by the moved instruction is live in the successor, the instruction is necessary. To handle the problems arising from upward motion, It can restrict the scheduler to prohibit it from moving an operation from its original block into a predecessor block. This avoids lengthening other paths that leave the predecessor block.

(A third choice exists. The compiler might make multiple copies of the block, depending on the surrounding context. In the example, B_5 could be cloned to provide two implementations, B_5' and B_5'', with the edge from B_2 connected to B_5' and the edge from B_3 connected to B_5''. This would create longer EBBs containing the new blocks. This kind of transformation is beyond the scope of the current discussion. See Section 14.1.8 for a discussion of cloning.)

To schedule an entire EBB, the compiler performs renaming, if necessary, over the region. Next, it builds a single precedence graph for the entire EBB, ignoring any exits from the EBB. It computes the priority metrics needed to select among ready operations and to break any ties. Finally, it applies the iterative scheduling scheme as with a single block. Each time that it assigns an operation to a specific cycle in the schedule, it must insert any compensation code required by that choice.

The compiler typically schedules each block once. Thus, in our example, the compiler might schedule (B_1, B_2, B_4) first, because it is the longest EBB. Choosing this EBB breaks up the EBB (B_1, B_3), so only (B_3) remains to be scheduled. Thus, the compiler might next schedule (B_3), then (B_5), and finally, (B_6). If the compiler has reason to believe that one EBB executes more often than another, it should give preference to that EBB and schedule it first so that it remains intact for scheduling.

11.4.2 Trace Scheduling

Trace scheduling uses information about actual run-time behavior of the program to select regions for scheduling. It uses profile information gathered by running an instrumented version of the program to determine which blocks execute most frequently. From this information, it constructs a trace, or a path through the control-flow graph, representing the most frequently executed path.

Given a trace, the scheduler applies the list scheduling algorithm to the entire trace, in much the same way that EBB scheduling applies it to an EBB. With an arbitrary trace, one additional complication arises—the scheduler may need to insert compensation code at points where control-flow enters the trace (a merge point). If the scheduler has moved an operation upward across the merge point, it needs to copy those operations to the end of the predecessors to the merge that lie outside the trace.

To schedule the entire procedure, the trace scheduler constructs a trace and schedules it. It then removes the blocks in the trace from consideration, and selects the next most frequently executed trace. This trace is scheduled, with the requirement that it respect any constraints imposed by previously scheduled code. The process continues, picking a trace, scheduling it, and removing it from consideration, until all the blocks have been scheduled.

EBB scheduling can be considered a degenerate case of trace scheduling, where the trace is selected using some static approximation to profiling data.

11.4.3 Scheduling Loops

Because loops play a critical role in many computationally-intensive tasks, they have received a great deal of attention in the literature on compilation. EBB scheduling can handle a portion of the loop body,[4] but it cannot address latencies that "wrap around" from the bottom of the loop back to the top. Trace scheduling can address these wrap around problems, but do so by creating multiple copies of any block that appears more than once in the trace. These shortcomings have led to several techniques that directly address the problem of generating excellent schedules for the body of an innermost loop.

Specialized loop scheduling techniques make sense only when the default scheduler is unable to produce compact and efficient code for the loop. If the loop, after scheduling, has a body that contains no stalls, interlocks, or NOPs, using a specialized loop scheduler is unlikely to improve its performance. Similarly, if the body of the loop is long enough that the end-of-block effects are a tiny fraction of the running time of the loop, a specialized scheduler is unlikely to help.

Still, there are many small, computationally-intensive loops that benefit from loop scheduling. Typically, these loops have too few instructions relative to the length of their critical paths to keep the underlying hardware busy. The key to scheduling these loops is to wrap the loop around itself. With a loop folded once, the first cycle in the loop might issue instructions from the start of the i^{th} iteration and from the middle of the $i-1^{st}$ iteration. With the loop folded twice, it would execute three iterations concurrently. Thus, the first cycle in the loop body might execute an early instruction from the i^{th} iteration, an instruction in the middle third of the $i-1^{st}$ iteration, and an instruction from the final third of the $i-2^{nd}$ iteration. Loops folded in this manner are often called *pipelined* loops because of the resemblance between this software technique and the hardware notion of pipelining a functional unit. (In both, multiple operations execute concurrently.)

For a folded loop to execute correctly, it must first execute a *prolog* section that fills up the pipeline. If the loop has three iterations executing at once in its kernel, the prolog must execute the first two thirds of iteration one, and the first third of iteration two. After the loop kernel completes, a corresponding *epilog* is needed to unwind the final iterations and complete them. The need for separate prolog and epilog sections increases code size. While the specific

[4] If the loop contains only one block, it does not need an extended basic block. If it contains any internal control-flow except for `break`-style jumps to the loop bottom, an EBB cannot include all of its blocks.

Cycle		*Functional Unit 0*				*Comments*
-4		loadI	@x	$\Rightarrow$	r$_{@x}$	*Set up the loop*
-3		loadI	@y	$\Rightarrow$	r$_{@y}$	*with initial loads*
-2		loadI	@z	$\Rightarrow$	r$_{@z}$	
-1		addI	r$_x$,792	$\Rightarrow$	r$_{ub}$	
1	L$_1$:	loadAO	r$_{sp}$,r$_{@x}$	$\Rightarrow$	r$_x$	*get* x[i] *&* y[i]
2		loadAO	r$_{sp}$,r$_{@y}$	$\Rightarrow$	r$_y$	
3		addI	r$_{@x}$, 4	$\Rightarrow$	r$_{@x}$	*bump the pointers*
4		addI	r$_{@y}$, 4	$\Rightarrow$	r$_{@y}$	*in shadow of loads*
5		mult	r$_x$,r$_y$	$\Rightarrow$	r$_z$	*the real work*
6		cmp_LT	r$_{@x}$,r$_{ub}$	$\Rightarrow$	r$_{cc}$	*shadow of mult*
7		store	r$_z$	$\Rightarrow$	r$_{sp}$,r$_{@z}$	*save the result*
8		addI	r$_{@z}$, 4	$\Rightarrow$	r$_{@z}$	*bump* z*'s pointer*
9		cbr	r$_{cc}$	$\rightarrow$	L$_1$,L$_2$	*loop-closing branch*

Figure 11.7: Example loop scheduled for one functional unit

increase is a function of the loop and the number of iterations that the kernel executes concurrently, it is not unusual for the prolog and epilog to double the amount of code required for the pipelined loop.

An example will make these ideas more concrete. Consider the following loop, written in C:

```
for (i=1; i < 200; i++)
    z[i] = x[i] * y[i];
```

Figure 11.7 shows the code that a compiler might generate for this loop, after some optimization. The figure shows the loop scheduled for a single functional unit. The cycle counts, in the first column, have been normalized to the first operation in the loop (at label L$_1$).

To generate this code, the compiler performed *operator strength reduction* which transformed the address computations into simple additions by the size of the array elements (four in this case). It also replaced the loop-ending test on i with an equivalent test on the address generated for x's next element, using a transformation called *linear function test replacement.* The last element of x accessed should be the 199th element, for an offset of $(199 - 1) \times 4 = 792$.

The pre-loop code initializes a pointer for each array and computes an upper bound for the range of r$_{@x}$ into r$_{ub}$. The loop uses r$_{ub}$ in cycle six to test for the end of the loop. The loop body loads x and y, performs the multiply, and stores the result into z. It increments the pointers to x, y, and z during the latency of the various memory operations. The comparison fills the cycle after the multiply, and the branch fills the final cycle of the store's latency.

Figure 11.8 shows the loop scheduled for a machine with two functional units. It assumes a target machine with two functional units, and relies on the strict ordering of ILOC operations. All operands are read at the start of the cycle in

Cycle		*Functional Unit 0*				*Functional Unit 1*			
-2		loadI	@x	$\Rightarrow$	$r_{@x}$	loadI	@y	$\Rightarrow$	$r_{@y}$
-1		loadI	@z	$\Rightarrow$	$r_{@z}$	addI	r_x,792	$\Rightarrow$	r_{ub}
1	L_1:	loadAO	r_{sp},$r_{@x}$	$\Rightarrow$	r_x	addI	$r_{@x}$, 4	$\Rightarrow$	$r_{@x}$
2		loadAO	r_{sp},$r_{@y}$	$\Rightarrow$	r_y	addI	$r_{@y}$, 4	$\Rightarrow$	$r_{@y}$
5		mult	r_x,r_y	$\Rightarrow$	r_z	cmp_LT	$r_{@x}$,r_{ub}	$\Rightarrow$	r_{cc}
7		store	r_z	$\Rightarrow$	r_{sp},$r_{@z}$	addI	$r_{@z}$, 4	$\Rightarrow$	$r_{@z}$
8		cbr	r_{cc}	$\rightarrow$	L_1,L_2	nop			

Figure 11.8: Example loop scheduled for two functional units

which an operation issues, and all definitions are performed at the end of the cycle in which the operation completes. Loads and stores must execute on unit zero, a typical restriction. The pre-loop code requires two cycles, as compared to four with a single functional unit. The loop body takes eight cycles; when the loop branches back to L_1, the store operation is still executing. (On many machines, the hardware would stall the loads until the store completed, adding another cycle to the loop 's execution time.)

With two functional units, the loop runs in either eight or nine cycles, depending on how the hardware handles the interaction between the store to z and the load from x. The presence of a second functional unit halved the time for the pre-loop code, but did little or nothing for the loop body itself. The scheduled loop still spends a significant portion of its time waiting for loads, stores, and multiplies to finish. This makes it a candidate for a more aggressive the loop scheduling techniques like pipelining.

Figure 11.9 shows the same loop after the compiler has pipelined it. The scheduler folded the loop once, so the kernel works on two distinct iterations concurrently. The prolog section starts up the pipeline by executing the first half of the first iteration. Each iteration of the kernel executes the second half of the i^{th} iteration and the first half of the $i+1^{st}$ iteration. The kernel terminates after processing the first half of the final iteration, so the epilog finishes the job by performing the second half of the final iteration.

On first examination, the relationship between the code in Figure 11.9 and the original loop is hard to see. Consider the pipelined loop on a line-by-line basis, referring back to Figure 11.8 for reference. All references to specific lines use the cycle number from the leftmost column of Figure 11.9.

The loop prolog To start the loop's execution, the code must perform all the operations in the pre-loop sequence of the original, along with the first half of iteration number one.

-4: The loadIs are from the pre-loop code. In the non-pipelined loop, they execute in cycle -2 of the pre-loop code.

-3: This is the first instruction of the first iteration. In the non-pipelined loop, it executes as cycle 1 of the loop body.

Cycle	*Functional Unit 0*					*Functional Unit 1*			
-4		loadI	@x	⇒	$r_{@x}$	loadI	@y	⇒	$r_{@y}$
-3		loadAO	r_{sp},$r_{@x}$	⇒	r_x	addI	$r_{@x}$, 4	⇒	$r_{@x}$
-2		loadAO	r_{sp},$r_{@y}$	⇒	r_y	addI	$r_{@y}$, 4	⇒	$r_{@y}$
-1		loadI	@z	⇒	$r_{@z}$	addI	r_x,788	⇒	r_{ub}
1	L_1:	loadAO	r_{sp},$r_{@x}$	⇒	r_x	addI	$r_{@x}$, 4	⇒	$r_{@x}$
2		loadAO	r_{sp},$r_{@y}$	⇒	r_y	mult	$r_{x'}$,r_y	⇒	r_z
3		i2i	r_x	⇒	$r_{x'}$	addI	$r_{@y}$, 4	⇒	$r_{@y}$
4		storeAO	r_z	⇒	r_{sp},$r_{@z}$	cmp_LT	$r_{@x}$,r_{ub}	⇒	r_{cc}
5		cbr	r_{cc}	→	L_1,L_2	addI	$r_{@z}$, 4	⇒	$r_{@z}$
+1	L_2:	mult	$r_{x'}$,r_y	⇒	r_z	nop			
+3		store	r_z	⇒	r_{sp},$r_{@z}$	nop			

Figure 11.9: Example loop after pipeline scheduling

-2: This is the second instruction of the first iteration. In the non-pipelined loop, it executes as cycle 2 of the loop body.

-1: These are the other two operations of the pre-loop code. In the non-pipelined loop, they form the instruction at cycle -1.

At the end of the prolog, the code has executed all of the code from the pre-loop sequence of the original loop, plus the first two instructions from the first iteration. This "fills the pipeline" so that the kernel can begin execution.

The loop kernel Each trip through the loop kernel executes the second half of iteration i, alongside the first half of the $i + 1^{st}$ iteration.

1: In this instruction, the `loadAO` is from iteration $i + 1$. It fetches `x[i+1]`. The `addI` increments the pointer for `x` so that it points to `x[i+2]`. The functional unit reads the value of $r_{@x}$ at the start of the cycle, and stores its new value back at the end of the iteration.

2: In this instruction, the `loadAO` is from iteration $i + 1$, while the `mult` is from iteration i. The `mult` cannot be scheduled earlier because of the latency of the load for `y` in the prolog (at cycle -2).

3: In this instruction, the `addI` is from iteration $i+1$; it updates the pointer to `y`. The `i2i` operation is new. It copies the value of `x` into another register so that it will not be overwritten before its use in the `mult` operation during the next kernel operation. (No copy is needed for `y` because the `loadAO` for `y` issues in the same cycle as the `mult` that uses its old value.

4: In this instruction, the `storeAO` is from iteration i while the `cmp_LT` is from iteration $i + 1$. The upper bound on `x`'s pointer, stored in r_{ub} has been adjusted to terminate the kernel after iteration 198, where the unpipelined loop terminated after iteration 199. The `storeAO` is scheduled as soon after the corresponding `mult` as possible, given the latency of the multiply.

5: In this instruction, the conditional branch is from iteration i, as is the `addI` that increments the pointer to `z`.

When the kernel finishes execution and branches to $\texttt{L}_2$, all the iterations have executed, except for the second half of the final iteration.

The loop epilog The epilog unwinds the final iteration. Because the scheduler knows that this code implements the last iteration, it can elide the update to $\texttt{r}_{@z}$ and the comparison and branch instructions. (Of course, if the code uses $\texttt{r}_{@z}$ after the loop, the scheduler will discover that it is `Live` on exit and will include the final increment operation.)

+1: This instruction executes the final multiply operation. Because the scheduler has eliminated the remaining updates and compares, the other operation is a `nop`.

+3: This instruction executes the final store. Again, the second operation is shown as a `nop`.

If the scheduler had more context surrounding the loop, it might schedule these epilog operations into the basic block that follows the loop. This would allow productive operations to replace the two `nop`s and to fill cycle *+2*.

Review In essence, the pipeline scheduler folded the loop in half to reduce the number of cycles spent per iteration. The kernel executes one fewer time than the original loop. Both the prolog and the epilog perform one half of an iteration. The pipelined loop uses many fewer cycles to execute the same operations, by overlapping the multiply and store from iteration i with the loads from iteration $i + 1$.

In general, the algorithms for producing pipelined loops require analysis techniques that are beyond the scope of this chapter. The curious reader should consult a textbook on advanced optimization techniques for details.

11.5 More Aggressive Techniques

Architectural developments in the 1990s increased the importance of instruction scheduling. The widespread use of processors with multiple functional units, increased use of pipelining, and growing memory latencies combined to make realized performance more dependent on execution order. This led to development of more aggressive approaches to scheduling. We will discuss two such approaches.

11.5.1 Cloning for Context

In the example from Figure 11.6, two of the EBBs contain more than one block—(B_1, B_2, B_4) and (B_1, B_3). When the compiler schedules (B_1, B_2, B_4) it must

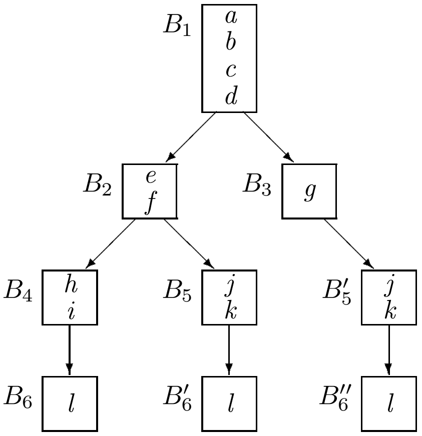


Figure 11.10: Cloning to Increase Scheduling Context

split (B_1, B_3) in half. B_1 is scheduled into the larger EBB, leaving B_3 to be scheduled as a single block. The other two EBBs in the example consist of single blocks as well—(B_5) and (B_6). Thus, any benefits derived from EBB scheduling occur only on the path B_1, B_2, B_4. Any other path through the fragment encounters code that has been subjected to local list scheduling.

If the compiler chose correctly, and B_1, B_2, B_4 is the most executed path, then splitting (B_1, B_3) is appropriate. If, instead, another path executes much more frequently, such as (B_1, B_3, B_5, B_6), then the benefits of EBB scheduling have been wasted. To increase the number of multiple-block EBBs that can be scheduled, the compiler may clone basic blocks to recapture context. (Cloning also decreases the performance penalty when the compiler's choice of an execution path is wrong.) With cloning, the compiler replicates blocks that have multiple predecessors to create longer EBBs.

Figure 11.10 shows the result of performing this kind of cloning on the example from Figure 11.6. Block B_5 has been cloned to create separate instances for the path from B_2 and the path from B_3. Similarly, B_6 has been cloned twice, to create an instance for each path entering it. This produces a graph with longer EBBs.

If the compiler still believes that (B_1, B_2, B_4) is the hot path, it will schedule (B_1, B_2, B_4, B_6) as a single EBB. This leaves two other EBBs to schedule. It will schedule (B_5, B'_6), using the result of scheduling (B_1, B_2) as context. It will schedule (B_3, B'_5, B''_6), using (B_1) as context. Contrast that with the simple EBB scheduler, which scheduled (B_3), (B_5), and (B_6) separately. In that scheme, only (B_3) could take advantage of any contextual knowledge about the code that executed before it, since both (B_5) and (B_6) have multiple predecessors and, therefore, inconsistent context. This extra context cost a second copy of statements j and k and two extra copies of statement l.

Of course, the compiler writer must place some limits on this style of cloning,

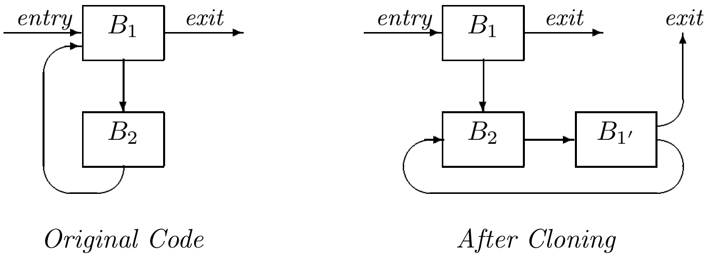


Figure 11.11: Cloning a Tail-call

to avoid excessive code growth and to avoid unwinding loops. A typical implementation might clone blocks within an innermost loop, stopping when it reaches a loop-closing edge (sometimes called a back-edge in the control-flow graph). This creates a situation where the only multiple-entry block in the loop is the first block in the loop. All other paths have only single-entry blocks.

A second example that merits consideration arises in tail-recursive programs. Recall, from Section 8.8.2, that a program is tail recursive if its last action is a recursive self-invocation. When the compiler detects a tail-call, it can convert the call into a branch back to the procedure's entry. This implements the recursion with an *ad hoc* looping construct rather than a full-weight procedure call. From the scheduler's point of view, however, cloning may improve the situation.

The left side of Figure 11.11 shows an abstracted control-flow graph for a tail-recursive routine, after the tail-call has been converted to iteration. Block B_1 is entered along two paths, the path from the procedure entry and the path from B_2. This forces the scheduler to use worst-case assumptions about what precedes B_1. By cloning B_1, as shown on the right, the compiler can create the situation where control enters $B_{1'}$ along only one edge. This may improve the results of regional scheduling, with an EBB scheduler or a loop scheduler. To further simplify the situation, the compiler might coalesce $B_{1'}$ onto the end of B_2, creating a single block loop body. The resulting loop can be scheduled with either a local scheduler or a loop scheduler, as appropriate.

11.5.2 Global Scheduling

Of course, the compiler could attempt to take a global approach to scheduling, just as compilers take a global approach to register allocation. Global scheduling schemes are a cross between code motion, performed quite late, and regional scheduling to handle the details of instruction ordering. They require a global graph to represent precedence; it must account for the flow of data along all possible control-flow paths in the program.

These algorithms typically determine, for each operation, the earliest position in the control-flow graph that is consistent with the constraints represented in the precedence graph. Given that location, they may move the operation later in the control-flow graph to the deepest location that is controlled by the same

Digression: Measuring Run-time Performance

The primary goal of instruction scheduling is to improve the running time of the generated code. Discussions of performance use many different metrics; the two most common are:

Operations per second The metric commonly used to advertise computers and to compare system performance is the number of operations executed in a second. This can be measured as instructions issued per second or instructions retired per second.

Time to complete a fixed task This metric uses one or more programs whose behavior is known, and compares the time required to complete these fixed tasks. This approach, called benchmarking, provides information about overall system performance, both hardware and software, on a particular workload.

No single metric contains enough information to allow evaluation of the quality of code generated by the compiler's back end. For example, if the measure is operations per second, does the compiler get extra credit for leaving extraneous (but independent) operations in code? The simple timing metric provides no information about what is achievable for the program. Thus, it allows one compiler to do better than another, but fails to show the distance between the generated code and what is optimal for that code on the target machine.

Numbers that the compiler writer might want to measure include the percentage of executed instructions whose output is actually used, and the percentage of cycles spent in stalls and interlocks. The former gives insight into some aspects of predicated execution, while the latter directly measures some aspects of schedule quality.

set of conditions. The former heuristic is intended to move operations out of deeply nested loops and into less frequently executed positions. Moving the operation earlier in the control-flow graph often increases, at least temporarily, the size of its live range. That leads to the latter heuristic, which tries to move the operation as close to its subsequent use without moving it any deeper in the nesting of the control-flow graph.

The compiler writer may be able to achieve similar results in a simpler way—using a specialized algorithm to perform code motion (such as Lazy Code Motion, described in Chapter 14) followed by a strong local or regional scheduler. The arguments for global optimization and for global allocation may not carry over to scheduling; the emphasis on scheduling is to avoid stalls, interlocks, and `nop`s. These latter issues tend to be localized in their impact.

11.6 Summary and Perspective

Algorithms that guarantee optimal schedules exist for simplified 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 [47]. It can be adapted to produce good code for expression DAGs. Fischer and Proebsting built on the labelling algorithm to derive an algorithm that produces optimal or near optimal results for small memory latencies [43]. Unfortunately, its has trouble when either the number of functional units or their latencies rise.

In practice, modern computers have become complex enough that none of the simplified models adequately reflect their behavior. Thus, most compilers use some form of list scheduling. The algorithm is easily adapted and parameterized; it can be run for forward scheduling and backward scheduling. The technology of list scheduling is the base from which more complex schedulers, like software pipeliners, are built.

Techniques that operate over larger regions have grown up in response to real problems. Trace scheduling was developed for VLIW architectures, where the compiler needed to keep many functional units busy. Techniques that schedule extended basic blocks and loops are, in essence, responses to the increase in both the number of pipelines that the compiler must consider and their individual latencies. As machines have become more complex, schedulers have needed a larger scheduling context to discover enough instruction-level parallelism to keep the machines busy.

The example for backward versus forward scheduling in Figure 11.4 was brought to our attention by Philip Schielke [46]. It is from the SPEC benchmark program go. It captures, concisely, an effect that has caused many compiler writers to include both forward and backward schedulers in their back ends.

Appendix A

ILOC

Introduction

Iloc is the linear assembly code for a simple abstract machine. The ILOC abstract machine is a RISC-like architecture. We have assumed, for simplicity, that all operations work on the same type of data—sixty-four bit integer data.

It has an unlimited number of registers. It has a couple of simple addressing modes, load and store operations, and three address register-to-register operators.

An ILOC program consists of a sequential list of instructions. An instruction may have a label; a label is followed immediately by a colon. If more than one label is needed, we represent it in writing by adding the special instruction NOP that performs no action. Formally:

$$\begin{array}{lcl}
\textit{iloc-program} & \rightarrow & \textit{instruction-list} \\
\textit{instruction-list} & \rightarrow & \textit{instruction} \\
 & | & \underline{\texttt{label}}\ \underline{:}\ \textit{instruction} \\
 & | & \textit{instruction instruction-list}
\end{array}$$

Each instruction contains one or more operations. A single-operation instruction is written on a line of its own, while a multi-operation instruction can span several lines. To group operations into a single instruction, we enclose them in square brackets and separate them with semi-colons. More formally:

$$\begin{array}{lcl}
\textit{instruction} & \rightarrow & \textit{operation} \\
 & | & \underline{[}\ \textit{operation-list}\ \underline{]} \\
\textit{operation-list} & \rightarrow & \textit{operation} \\
 & | & \textit{operation}\ \underline{;}\ \textit{operation-list}
\end{array}$$

An ILOC operation corresponds to an instruction that might be issued to a single functional unit in a single cycle. It has an optional label, an opcode, a set of source operands, and a set of target operands. The sources are separated from the targets by the symbol ⇒, pronounced “into.”

$$\begin{array}{lcl}
\textit{operation} & \rightarrow & \textit{opcode operand-list} \; \underline{\Rightarrow} \; \textit{operand-list} \\
\textit{operand-list} & \rightarrow & \textit{operand} \\
 & | & \textit{operand} \; \underline{,} \; \textit{operand-list} \\
\textit{operand} & \rightarrow & \underline{\texttt{register}} \\
 & | & \underline{\texttt{number}} \\
 & | & \underline{\texttt{label}}
\end{array}$$

Operands come in three types: `register`, `number`, and `label`. The type of each operand is determined by the opcode and the position of the operand in the operation. In the examples, we make this textually obvious by beginning all `register` operands with the letter `r` and all `label`s with a letter other than `r` (typically, with an `l`).

We assume that source operands are read at the beginning of the cycle when the operation issues and that target operands are defined at the end of the cycle in which the operation completes.

Most operations have a single target operand; some of the `store` operations have multiple target operations. For example, the `storeAI` operation has a single source operand and two target operands. The source must be a register, and the targets must be a register and an immediate constant. Thus, the ILOC operation

$$\texttt{storeAI r}_i \Rightarrow \texttt{r}_j\texttt{, 4}$$

computes an address by adding 4 to the contents of $\texttt{r}_j$ and stores the value found in $\texttt{r}_i$ into the memory location specified by the address. In other words,

$$\textsc{Memory}(\texttt{r}_j\texttt{+4}) \leftarrow \textsc{Contents}(\texttt{r}_i)$$

The non-terminal *opcode* can be any of the ILOC operation codes. Unfortunately, as in a real assembly language, the relationship between an *opcode* and the form of its arguments is less than systematic. The easiest way to specify the form of each opcode is in a tabular form. Figure A.2 at the end of this appendix shows the number of operands and their types for each ILOC opcode used in the book.

As a lexical matter, ILOC comments begin with the string `//` and continue until the end of a line. We assume that these are stripped out by the scanner; thus they can occur anywhere in an instruction and are not mentioned in the grammar.

To make this discussion more concrete, let's work through the example used in Chapter 1. It is shown in Figure A.1. To start, notice the comments on the right edge of most lines. In our ILOC-based systems, comments are automatically generated by the compiler's front end, to make the ILOC code more readable by humans. Since the examples in this book are intended primarily for humans, we continue this tradition of annotating the ILOC

This example assumes that register $\texttt{r}_{sp}$ holds the address of a region in memory where the variables `w`, `x`, `y`, and `z` are stored at offsets 0, 8, 16, and 24, respectively.

```
loadAI   r_sp, 0   => r_w       // w is at offset 0 from r_sp
loadI    2         => r_2       // constant 2 into r_2
loadAI   r_sp, 8   => r_x       // x is at offset 8
loadAI   r_sp, 12  => r_y       // y is at offset 12
loadAI   r_sp, 16  => r_z       // z is at offset 16
mult     r_w, r_2  => r_w       // r_w <- w×2
mult     r_w, r_x  => r_w       // r_w <- (w×2) × x
mult     r_w, r_y  => r_w       // r_w <- (w×2×x) × y
mult     r_w, r_z  => r_w       // r_w <- (w×2×x×y) × z
storeAI  r_w       => r_sp, 0   // write r_w back to 'w'
```

Figure A.1: Introductory example, revisited

The first instruction is a loadAI operation, or a *load address-immediate*. From the opcode table, we can see that it combines the contents of r_{sp} with the immediate constant 0 and retrieves the value found in memory at that address. We know, from above, that this value is w. It stores the retrieved value in r_w. The next instruction is a loadi operation, or a *load immediate*. It moves the value 2 directly into r_2. (Effectively, it reads a constant out of the instruction stream and into a register.) Instructions three through five load the values of x into r_x, y into r_y, and z into r_z.

The sixth instruction multiplies the contents of r_w and r_2, storing the result back into r_w. Instruction seven multiplies this quantity by r_x. Instruction eight multiplies in r_y, and instruction nine picks up r_z. In each instruction from six through nine, the value is accumulated into r_w.

Finally, instruction ten saves the value to memory. It uses a storeAI, or *store address-immediate*, to write the contents of r_w into the memory location at offset 0 from r_{sp}. As pointed out in Chapter 1, this sequence evaluates the expression

$$\mathtt{w} \leftarrow \mathtt{w} \times 2 \times \mathtt{x} \times \mathtt{y} \times \mathtt{z}$$

The opcode table, at the end of this appendix, lists all of the opcodes used in ILOC examples in the book.

Comparison and Conditional Branch In general, the ILOC comparison operators take two values and return a boolean value. The operations cmp_LT, cmp_LE, cmp_EQ, cmp_NE, cmp_GE, and cmp_GT work this way. The corresponding conditional branch, cbr, takes a boolean as its argument and transfers control to one of two target labels. The first label is selected if the boolean is true; the second label is selected if the label is false.

Using two labels on the conditional branch has two advantages. First, the code is somewhat more concise. In several situations, a conditional branch might be followed by an absolute branch. The two-label branch lets us record that combination in a single operation. Second, the code is easier to manipulate. A single-label conditional branch implies some positional relationship with the next instruction; there is an implicit connection between the branch and its "fall-through" path. The compiler must take care, particularly when reading and writing linear code, to preserve these relationships. The two-label conditional branch makes this implicit connection explicit, and removes any possible positional dependence.

Because the two branch targets are not "defined" by the instruction, we change the syntax slightly. Rather than use the ⇒ arrow, we write branches with the smaller → arrow.

In a few places, we want to discuss what happens when the comparison returns a complex value written into a designated area for a "condition code." The condition code is a multi-bit value that can only be interpreted with a more complex conditional branch instruction. To talk about this mechanism, we use an alternate set of comparison and conditional branch operators. The comparison operator, `comp` takes two values and sets the condition code appropriately. We always designate the target of `comp` as a condition code register by writing it $\texttt{cc}_i$. The corresponding conditional branch has six variants, one for each comparison result. Figure A.3 shows these instructions and their meaning.

Note: In a real compiler that used ILOC, we would need to introduce some representation for distinct data types. The research compiler that we built using ILOC had several distinct data types—integer, single-precision floating-point, double-precision floating-point, complex, and pointer.

A.0.1 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 $\texttt{r}_{1776}$.

3. The register $\texttt{r}_0$ is reserved as a pointer the current activation record. Thus, the operation `loadAI` $\texttt{r}_0$,`@x` ⇒$\texttt{r}_1$ implicitly exposes the fact that the variable `x` is stored in the activation record of the procedure containing the operation.

A.0.2 Other Important Points

An ILOC operation reads its operands at the start of the cycle in which it issues. It writes its target at the end of the cycle in which it finishes. Thus, two operations in the same instruction can both refer to a given register. Any uses receive the value defined at the start of the cycle. Any definitions occur at the end of the cycle. This is particularly important in Figure 11.9.

Opcode	*Sources*	*Targets*	*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	$r_1 \div r_2 \rightarrow r_3$
addI	r_1, c_1	r_2	$r_1 + c_1 \rightarrow r_2$
subI	r_1, c_1	r_2	$r_1 - c_1 \rightarrow r_2$
multI	r_1, c_1	r_2	$r_1 \times c_1 \rightarrow r_2$
divI	r_1, c_1	r_2	$r_1 \div c_1 \rightarrow r_2$
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	$r_1 \gg c_2 \rightarrow r_3$
loadI	c_1	r_2	$c_1 \rightarrow r_2$
load	r_1	r_2	$\textsc{Memory}(r_1) \rightarrow r_2$
loadAI	r_1, c_1	r_2	$\textsc{Memory}(r_1+c_1) \rightarrow r_2$
loadAO	r_1, r_2	r_3	$\textsc{Memory}(r_1+r_2) \rightarrow r_3$
cload	r_1	r_2	character load
cloadAI	r_1, r_2	r_3	character loadAI
cloadAO	r_1, r_2	r_3	character loadAO
store	r_1	r_2	$r_1 \rightarrow \textsc{Memory}(r_2)$
storeAI	r_1	r_2, c_1	$r_1 \rightarrow \textsc{Memory}(r_2+c_1)$
storeAO	r_1	r_2, r_3	$r_1 \rightarrow \textsc{Memory}(r_2+r_3)$
cstore	r_1	r_2	character store
cstoreAI	r_1	r_2, r_3	character storeAI
cstoreAO	r_1	r_2, r_3	character storeAO
br		l_1	$l_1 \rightarrow$ PC
cbr	r_1	l_1, l_2	r_1 = true $\Rightarrow l_1 \rightarrow$ PC r_1 = false $\Rightarrow l_2 \rightarrow$ PC
cmp_LT	r_1, r_2	r_3	$r_1 < r_2 \Rightarrow$ true $\rightarrow r_3$ (otherwise, false $\rightarrow r_3$)
cmp_LE	r_1, r_2	r_3	$r_1 \leq r_2 \Rightarrow$ true $\rightarrow r_3$
cmp_EQ	r_1, r_2	r_3	$r_1 = r_2 \Rightarrow$ true $\rightarrow r_3$
cmp_NE	r_1, r_2	r_3	$r_1 \neq r_2 \Rightarrow$ true $\rightarrow r_3$
cmp_GE	r_1, r_2	r_3	$r_1 \geq r_2 \Rightarrow$ true $\rightarrow r_3$
cmp_GT	r_1, r_2	r_3	$r_1 > r_2 \Rightarrow$ true $\rightarrow r_3$
i2i	r_1	r_2	$r_1 \rightarrow r_2$
c2c	r_1	r_2	$r_1 \rightarrow r_2$
c2i	r_1	r_2	converts character to integer
i2c	r_1	r_2	converts integer to character

Figure A.2: Iloc opcode table

Opcode	*Sources*	*Targets*	*Meaning*
comp	r_1, r_2	cc_1	*defines* cc_1
cbr_LT	cc_1	l_1, l_2	cc_1 = LT ⇒ l_1 → PC (otherwise l_2 → PC)
cbr_LE	cc_1	l_1, l_2	cc_1 = LE ⇒ l_1 → PC
cbr_EQ	cc_1	l_1, l_2	cc_1 = EQ ⇒ l_1 → PC
cbr_GE	cc_1	l_1, l_2	cc_1 = GE ⇒ l_1 → PC
cbr_GT	cc_1	l_1, l_2	cc_1 = GT ⇒ l_1 → PC
cbr_NE	cc_1	l_1, l_2	cc_1 = NE ⇒ l_1 → PC

Figure A.3: Alternate Compare/Branch Syntax

Appendix B

Data Structures

B.1 Introduction

Crafting a successful compiler requires attention to many details. This appendix explores minor design details that have an impact on a compiler's implementation. In most cases, the details would burden the discussion in the body of the book. Thus, we have gathered them together into this appendix.

B.2 Representing Sets

B.3 Implementing Intermediate Forms

B.4 Implementing Hash-tables

The two central problems in hash-table implementation are ensuring that the hash function produces a good distribution of integers (at any of the table sizes that will be used), and providing conflict resolution in an efficient way.

Finding good hash functions is difficult. Fortunately, hashing has been in use long enough that many good functions have been described in the literature. Section B.4.1 describes two related approaches that, in practice, produce good results.

The problem of collision handling drives the layout of hash tables. Many collision-resolution schemes have been proposed. We will discuss the two most widely used schemes. Section B.4.2 describes open hashing, sometimes called a bucket hash, while Section B.4.3 presents an alternative scheme called open addressing, or rehashing. Section B.4.5 shows how to incorporate the mechanisms for lexical scoping into these schemes. The final section deals with a practical issue that arises in a compiler development environment—frequent changes to the hash-table definition.

Digression: Organizing a Search Table

In designing a symbol table, the first decision that the compiler writer faces concerns the organization of the table and its searching algorithm. As in many other applications, the compiler writer has several choices.

- Linear List: A linear list can expand to arbitrary size. The search algorithm is a single, small, tight loop. Unfortunately, the search algorithm requires $\mathsf{O}(n)$ probes per lookup, on average. This single disadvantage probably outweighs simplicity of implementation and expansion. To justify using a linear list, the compiler writer needs strong evidence that the procedures being compiled have very few names.
- Binary Search: To retain the easy expansion of the linear list while improving search time, the compiler writer might use a balanced binary tree. Ideally, a binary tree should allow lookup in $\mathsf{O}(\log_2 n)$ probes per lookup; this is a considerable improvement over the linear list. Many algorithms have been published for balancing search trees [27, 12]. (Similar effects can be achieved by using a binary search of an ordered table, but the table makes insertion and expansion more difficult.)
- Hash Table: A hash table may minimize access costs. The implementation computes a table index directly from the name. As long as that computation produces a good distribution of indices, the average access cost should be $\mathsf{O}(1)$. Worst-case, however, can devolve to linear search. The compiler writer can take steps to decrease the likelihood of this happening, but pathological cases may still occur. Many hash-table implementations have inexpensive schemes for expansion.
- Multi-set Discrimination: To avoid worst-case behavior, the compiler writer can use an off-line technique called multiset discrimination. It creates a unique index for each identifier, at the cost of an extra pass over the source text. This technique avoids the possibility of pathological behavior that always exists with hashing. (See the digression “An Alternative to Hashing” on page 156 for more details.)

Of these organizations, the most common choice appears to be the hash table. It provides better compile-time behavior than the linear list or binary tree, and the implementation techniques have been widely studied and taught.

B.4.1 Choosing a Hash Function

The importance of a good hash function cannot be overemphasized. A hash function that produces a bad distribution of index values directly increases the average cost of inserting items into the table and finding such items later. Fortunately, many good hash functions have been documented in the literature. The multiplicative hash function, described by Knuth and Cormen Cormen *et al.* [38, 27] works well and has efficient implementations. The notion of a universal hash function described by Cormen *et al.* [27] leads to a related hash function, but incorporates randomization to avoid pathological behavior.

Multiplicative Hash Function The multiplicative hash function is deceptively simple. The programmer chooses a single constant, C, and use it in the following formula

$$h(\mathit{key}) \;=\; \lfloor \mathit{TableSize}((C \cdot \mathit{key}) \bmod 1)\rfloor$$

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 computing C as an integer constant A divided by the wordsize of the computer. In our experience, C should be between The effect of the function is to compute $C \cdot \mathit{key}$, 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 can be parameterized by a small set of constants. At execution time, a set of 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. (The same constants are used throughout a single execution of the program that uses the hash function, but the constants vary from execution to execution.) By varying the hash function on each execution of the program, a universal hash function produces different distributions on each run of the program. In a compiler, if the input program produced pathological behavior in some particular compile, it should not produce the same behavior in subsequent compiles.

To implement a universal version of the multiplicative hash function, the programmer has two choices. The simplest strategy is to randomly generate an integer A at the beginning of execution, form $\frac{A}{w}$, and use that as C in the computation. The other alternative is to pick a set of ten to twenty integer constants A, encode them in a table, and, at run-time, select one constant at random for use in computing C (as $\frac{A}{w}$). The advantage of the latter method is that the programmer can screen constants for good behavior. In the former approach, the random number generator directly changes the distribution of keys across the table. In the latter approach, the random number generator only needs to be good enough to pick an integer index into the table—the keys in the table have been pre-selected for reasonable behavior.

Digression: The Perils of Poor Hash Functions

The choice of a hash function has a critical impact on the cost of table insertions and lookups. This is a case where a small amount of attention can make a large difference.

Many years ago, we saw a student implement the following hash function for character strings: (1) break the key in four byte chunks, (2) exclusive-or them together, and (3) take the resulting number, modulo table size, as the index. The function is relatively fast. It has a straight forward implementation. With some table sizes, it produces adequate distributions.

When the student inserted this implementation into a system that performed source-to-source translation on Fortran programs, several independent facts combined to create an algorithmic disaster. First, the implementation language padded character strings with blanks to reach a four-byte boundary. Second, the student chose an initial table size of 2048. Finally, Fortran programmers use many one and two character variable names, such as `i`, `j`, `k`, `x`, `y`, and `z`.

All the short variable names fit in a single word string, so no exclusive-or is needed. However, taking x mod 2048 simply masks out the final eleven bits of x. Thus, all short variable names produce the same index—the final bits of the string consisting of two blanks. Thus, the table devolved rapidly into a linear search. While this particular hash function is far from ideal, simply changing the table size to 2047 eliminated the most noticeable effects.

B.4.2 Open Hashing

Open hashing, also called bucket hashing, assumes that the hash function h will produce 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 $\mathsf{O}(\frac{N}{S})$. As long as h distributes names fairly uniformly and the ratio of buckets to names is kept small, this cost approximates our goal: $\mathsf{O}(1)$ time for each access.

Figure B.1 shows a small hash table implemented with this scheme. It assumes that $h(\mathtt{a}) = h(\mathtt{d}) = 3$ to create a collision. Thus, `a` and `d` occupy the same slot in the table. The list structure links them together and reveals that `a` was entered before `d`; after all, `Insert` should add to the front of the list rather than the end of the list.

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 buckets is usually an array of pointers, the overhead for

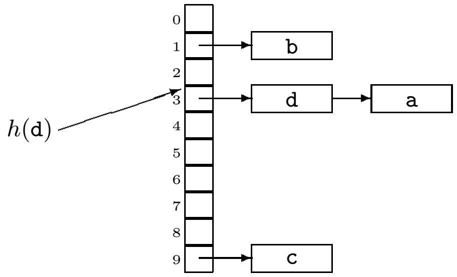


Figure B.1: Open hashing table

increasing S is small—one pointer for each added bucket. (This makes it less expensive to keep $\frac{N}{S}$ small. The cost per name is constant.) Choosing S as a power of two reduces the cost of the inevitable mod operation required to implement h.

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 heavyweight memory allocation, this may be noticeable. Using a lighter-weight mechanism, such as arena-based allocation (see the digression on 195) can alleviate this problem.

2. If any particular set gets large, `LookUp` degrades into linear search. With a reasonably behaved hash function, this only occurs when $N \gg S$. The implementation should detect this problem and enlarge the array of buckets. Typically, this involves allocating a new array of buckets and re-inserting each entry from the old table into the new table.

A well implemented open hash table provides efficient access with a low overhead in both space and time.

To improve the behavior of the linear search performed inside a single bucket, the compiler can dynamically reorder the chain. Rivest and Tarjan showed that the appropriate strategy is to move a node up the chain by one position on each lookup.

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 it is empty, `LookUp` fails. If it finds n, `LookUp` succeeds. If it finds a name other than $h(n)$, it uses a second function $g(n)$ to compute an increment for the search and probes the table at $(h(n)+g(n))$ mod S, $(h(n)+2\times g(n))\text{mod}S$, $(h(n)+3\times g(n))\text{mod}S$, and so on, until it either finds n, finds an empty slot, or returns to $h(n)$ a second time. If it finds an empty slot, or it returns to $h(n)$ a second time, `LookUp` fails.

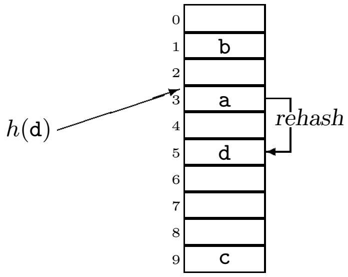


Figure B.2: Open Addressing Table

Figure B.2 shows a small hash table implemented with this scheme. It uses the same data as the table shown in Figure B.1. As before, $h(\texttt{a}) = h(\texttt{d}) = 3$, while $h(\texttt{b}) = 1$ and $h(\texttt{c}) = 9$. When `d` was inserted, it produced a collision with `a`. The secondary hash function $g(\texttt{d})$ produced 2, so `Insert` placed `d` at index 5 in the table. 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, and a single table location can serve as 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 collisions are infrequent, because h and g produce good distributions, then the rehash chains stay short and access costs stay low. Because it can recompute g inexpensively, it need not store pointers to form the rehash chains—a savings of N pointers. This saved space goes into making the table larger; the larger table improves behavior by lowering the collision frequency. 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 induce interfere with a subsequent insertion of 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 the scheme behaves well, 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. (Similarly, the implementation may expand S when some chain becomes too long.) Expansion is needed for correctness; with open hashing, it is a matter of efficiency.

Some implementations use a constant function for g. This simplifies the implementation and reduces the cost of computing secondary indices. However, it creates a single rehash chain for each value of h and it has the effect of merging rehash chains whenever a secondary index encounters an already occupied table slot. These two disadvantages outweigh the cost of evaluating a second hash

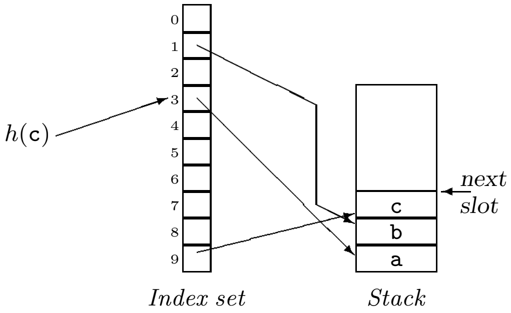


Figure B.3: Stack allocation for records

function. A more reasonable choice is to use two multiplicative hash functions with different constants—selected randomly at startup from a table of constants, if possible.

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 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 property 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. A small set of primes will suffices, 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 recorded information associated with each hash table key. With open hashing, the temptation is to allocate the records directly in the nodes that implement the chains. With open addressing, the temptation is to avoid the pointers and make each entry in the index table be a symbol record. Both these approaches have drawbacks. We may achieve better results by using a separately allocated stack to hold the records.

Figure B.3 depicts this implementation. In an open hashing implementation, the chain lists themselves can be implemented on the stack. This lowers the cost of allocating individual records—particularly if allocation is a heavyweight operation. In an open addressing implementation, the rehash chains are still implicit in the index set, preserving the space saving that motivated the idea.

In this design, the actual records form a dense table, better for external I/O. With heavyweight allocation, this scheme amortizes the cost of a large allocation over many records. With a garbage collector, it decreases the number of objects that must be marked and collected. In either case, having a dense table makes it more efficient to iterate over all of the symbols in the table—an operation

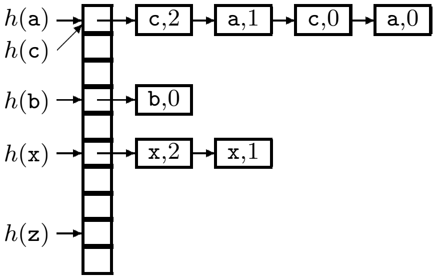


Figure B.4: Lexical Scoping in an Open Hashing Table

that the compiler will use to perform tasks such as assigning storage locations.

As a final advantage, this scheme drastically simplifies the task of expanding the index set. To expand the index set, the compiler discards the old index set, allocates a larger set, and then reinserts the records into the new table, working from the bottom of the stack to the top. This eliminates the need to have, temporarily, both the old and new table in memory. Iterating over the dense table takes less work, in general, than chasing the pointers to traverse the lists in open hashing. It avoids iterating over empty table slots, as can happen when open addressing expands the index set to keep chain lengths short.

The compiler need not allocate the entire stack as a single object. Instead, the stack can be implemented as a chain of nodes that each hold k records. When a node is full, the implementation allocates a new node, adds it to the end of the chain, and continues. This provides the compiler writer with fine-grain control over the tradeoff between allocation cost and wasted space.

B.4.5 Adding Nested Lexical Scopes

Section 6.7.3 described the issues that arise in creating a symbol table to handle nested lexical scopes. It described a simple implementation that created a sheaf of symbol tables, one per level. While that implementation is conceptually clean, it pushes the overhead of scoping into `LookUp`, rather than into `InitializeScope`, `FinalizeScope`, and `Insert`. Since the compiler invokes `LookUp` many more times than it invokes these other routines, other implementations deserve consideration.

Consider again the code in Figure 6.9. It generates the following actions:

$$\uparrow \langle \texttt{a},0\rangle \; \langle \texttt{b},0\rangle \; \langle \texttt{c},0\rangle \uparrow \langle \texttt{b},1\rangle \; \langle \texttt{z},1\rangle \downarrow \uparrow \langle \texttt{a},1\rangle \; \langle \texttt{x},1\rangle \uparrow \langle \texttt{c},2\rangle, \langle \texttt{x},2\rangle \downarrow \downarrow \downarrow$$

where ↑ shows a call to `InitializeScope()`, ↓ is a call to `FinalizeScope()`, and a pair $\langle \texttt{name},n\rangle$ is a call to `Insert` that adds `name` at level n.

Adding Lexical Scopes to Open Hashing Consider what would happen in an open-hashing table if we simply added a lexical-level field to the record for each name, and inserted new names at the front of its chain. `Insert` could check for duplicates by comparing both names and lexical levels. `LookUp` would return the first record that it discovered for a given name. `InitializeScope` simply

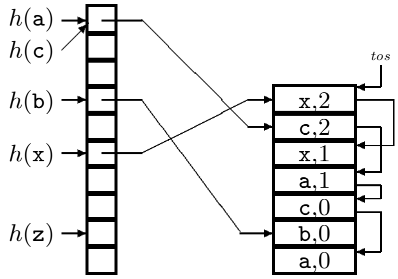


Figure B.5: Lexical Scoping in a Stack Allocated Open Hashing Table

bumps an internal counter for the current lexical level. This scheme pushes the complications into `FinalizeScope`, which must not only decrement the current lexical level, but also must remove the records for any names inserted in this scope.

If open hashing is implemented with individually allocated nodes for its chains, as shown in Figure B.1, then `FinalizeScope` must find each record for the scope being discarded and remove them from their respective chains. If they will not be used later in the compiler, it must deallocate them; otherwise, it must chain them together to preserve them. Figure B.4 shows the table that this approach would produce, at the assignment statement in Figure 6.9.

With stack allocated records, `FinalizeScope` can iterate from the top of the stack downward until it reaches a record for some level below the level being discarded. For each record, it updates the index set entry with the record's pointer to the next item on the chain. If the records are being discarded, `FinalizeScope` resets the pointer to the next available slot; otherwise, the records are preserved, together, on the stack. Figure B.5 shows the symbol table for our example, at the assignment statement.

With a little care, dynamic reordering of the chain can be added to this scheme. Since `FinalizeScope()` uses the stack ordering, rather than the chain ordering, it will still find all the top-level names at the top of the stack. With reordered chains, the compiler either needs to walk the chain to remove each deleted name record, or to doubly-link the chains to allow for quicker deletion.

Adding Lexical Scopes to Open Addressing With an open addressing table, the situation is slightly more complex. Slots in the table are a critical resource, when all the slots are filled, the table must be expanded before further insertion can occur. Deletion from a table that uses rehashing is difficult; the implementation cannot easily tell if the deleted record falls in the middle of some rehash chain. Thus, marking the slot empty breaks any chain that passes through that location (rather than ending there). This argues against storing discrete records for each variant of a name in the table. Instead, the compiler should link only one record per name into the table; it can create a chain of superseded records for older variants. Figure B.6 depicts this situation for the continuing example.

This scheme pushes most of the complexity into `Insert` and `FinalizeScope`.

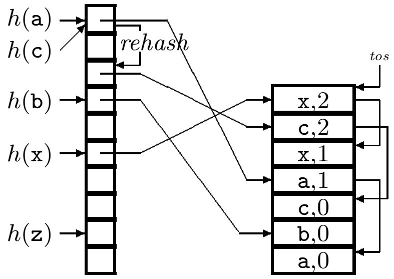


Figure B.6: Lexical Scoping in an Open Addressing Table

If Insert creates a new record on the top of the stack. If it finds an older declaration of the same in the index set, it replaces that reference with a reference to the new record, and links the older reference into the new record. FinalizeScope iterates over the top of the stack, as in open hashing. To remove a record that has an older variant, it simply relinks the index set to point at the older variant. To remove the final variant of a name, it must insert a reference to a specially-designated record that denotes a deleted reference. LookUp must recognize the deleted reference as occupying a slot in the current chain. Insert must know that it can replace a deleted reference with any newly inserted symbol.

This scheme, in essence, creates separate chains for collisions and for redeclarations. Collisions are threaded through the index set. Redeclarations are threaded through the stack. This should reduce the cost of LookUp slightly, since it avoids examining more than one record name for any single name.

Consider a bucket in open hashing that contains seven declarations for x and a single declaration for y at level zero. LookUp might encounter all seven records for x before finding y. With the open addressing scheme, LookUp encounters one record for x and one record for y.

B.5 Symbol Tables for Development Environments

Most compilers use a symbol table as a central repository for information about the various names that arise 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. Fields are added to communicate information between passes. When the need for a field disappears, it may or may not be removed from the symbol table definition. As each field is added, the symbol table swells in size and any parts of the compiler with direct access to the symbol table must be recompiled.

We encountered this problem in the implementation the $\mathcal{R}^n$ and ParaScope Programming Environments. The experimental nature of these systems led to a situation where additions and deletions of symbol table fields were common. To address the problem, we implemented more complex, but more flexible structure for the symbol table—a two-dimensional hash table. This eliminated almost all changes to the symbol table definition and its implementation.

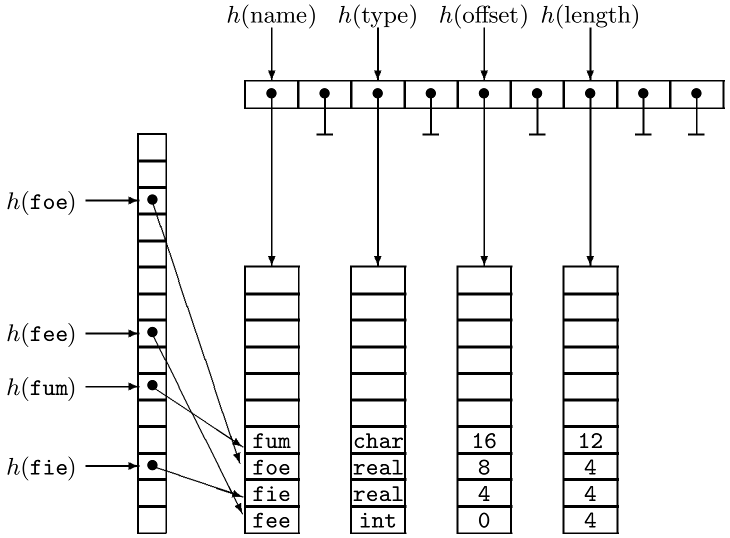


Figure B.7: Two-dimensional, hashed, symbol table

The two-dimensional table, shown in Figure B.7, uses two distinct hash index tables. The first, shown along the left edge of the drawing, corresponds to the sparse index table from Figure B.3. The implementation uses this table to hash on symbol name. The second, shown along the top of the drawing, is a hash table for field-names. The programmer references individual fields by both their textual name and the name of the symbol; the implementation hashes a the symbol name to obtain an index and the field name to select a vector of data. The desired attribute is stored in the vector under the symbol's index. It behaves as if each field has its own hash table, implemented as shown in Figure B.3.

While this 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 the field; this avoids the space overhead of unused fields. It allows individual developers to create and delete symbol table fields without interfering with other programmers.

Our 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. This scheme allows individual programmers to manage their own symbol table use in a responsible and independent way, without interference from other programmers and their code.

As a final issue, the implementation should be abstracted with respect to a specific symbol table. That is, it should always take a table instance as a parameter. This allows the compiler to reuse the implementation in many cases. For example, a well-written set of symbol table routines greatly simplifies the implementation of value numbering, using either the classic algorithm for single blocks or Simpson's SCC algorithm for entire procedures.

Appendix C

Abbreviations, Acronyms, and Glossary

The text follows common practice in Computer Science by taking long phrases, such as *world-wide web* and replacing them with abbreviations or acronyms, such as WWW. This simplifies writing; WWW is much shorter than *world-wide web*. Sometimes, it simplifies conversation. (Sometimes, it does not; for example, *world-wide web* has the misfortune of beginning with a letter whose name is, itself, two syllables. Thus, WWW has six syllables when *world-wide web* has three!).

This list combines abbreviations, acronyms, and terms that simply need an easily located definition.

AG attribute grammar

AR activation record

ARP activation record pointer – The activation record for the current procedure contains critical information that is needed for proper execution of the program. (See Chapters 7 and 8.) The ARP is the code's run-time pointer to the current activation record. Typically, it is kept in a designated register. The ILOC examples throughout the book assume that R_0 contains the ARP. (Some authors refer to the ARP as a "frame pointer.")

AST abstract syntax tree

CODE SHAPE The term "code shape" appears to have originated in the Bliss-11 compiler project at Carnegie-Mellon University [51]. The term is somewhat intuitive; it refers to the high-level decisions that go into choosing a particular instruction sequence to implement a fragment of source-language code.

CFG context-free grammar

CFL context-free language

CSL context-sensitive language

CSG context-sensitive grammar

DEFINITION the point in the code where a value is defined; in common use, it refers to the operation whose evaluation creates the value. (See also USE).

DEGREE used, with reference to a node in a graph, as the number of edges that connect to that node. In the text, the degree of a node n is sometimes denoted n°, using a notation due to the Babylonians.

DFA deterministic finite automaton

DG data dependence graph

DOPE VECTOR The vector of information passed at a call site to describe an array and its dimensions. The vector contains the "inside dope" on the array—a late fifties slang term.

FA finite automaton – used to refer, without bias, to either a DFA or an NFA

FP frame pointer – see ARP. The term "frame" refers to a "stack frame". Activation records that are kept on the run-time stack are sometimes called stack frames. This abbreviation should not occur in the text. If it does, please report it to `eac@cs.rice.edu`.

HEURISTIC a rule used to guide some complex process. Heuristic techniques are often used to approximate the solution of an instance of some NP-Complete problem.

A classic heuristic for whitewater canoeing is *"don't hit rocks and don't fall out of the boat."* The amateur who follows this heuristic improves the chances of survival for both canoeist and canoe.

IL intermediate language (or intermediate representation) See Chapter 6.

INSTRUCTION the collection of operations that are issued in the same cycle.

IR intermediate representation (or intermediate language) See Chapter 6.

ILOC Intermediate language for an optimizing compiler – the low-level intermediate language used in most of the examples in this book. See Appendix A).

INTERVAL GRAPH ⟨*need a good definition*⟩

LR(1) a popular, bottom-up, deterministic parsing technique. The name derives from the fact that the parser makes a *left-to right scan* while building a *reverse rightmost derivation* using a lookahead of at most 1 symbol.

NFA non-deterministic finite automaton

NP-Complete A specific class of problems in the hierarchy of computational complexity. It is an open question whether or not deterministic polynomial-time algorithms exist for these problems. If a deterministic polynomial-time algorithm exists for any of these problems, one exists for all of these problems. See a good text on algorithms, such as Cormen *et al.* [27] for a deeper discussion of this issue.

In practical, compiler-related terms, the only known algorithms that guarantee optimal solutions to instances of an NP-Complete problem require $\mathsf{O}(2^n)$ time, where n is some measure of the problem size. Compilers must approximate solutions to several NP-Complete problems, including the realistic versions of register allocation and instruction scheduling.

OOL object-oriented language

OOP object-oriented programming

OPERATION an indivisible unit of work performed by one functional unit on a modern processor. Examples include loading or storing a value, adding two registers, or taking a branch.

RE regular expression – a notation used to describe a regular language. See Section 2.2.

ROM read-only memory – used to describe several types of non-volatile memory where the contents are immutable. In common use, it refers to a semiconductor memory with predetermined contents. The term is also used to refer to the contents of a compact disc (CD-ROM) or a digital video disk (DVD-ROM).

TOS top of stack – For many languages, a stack discipline can be used to allocate activation records. (See Chapters 7 and 8.) The TOS is a pointer to the current location of the extendable end of the stack. If the compiler needs to generate code that extends the current AR, it needs a pointer to TOS in addition to the ARP.

USE a reference that uses some value, usually as an operand to some operation. In common use, the term refers to the point in the code where the reference occurs. (See also DEFINITION.)

Index

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling.* Prentice-HalL, Inc., Englewood Cliffs, New Jersey, 1973.

[2] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. *SIGPLAN Notices*, 32(6):134–145, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[3] J. 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, and R. Nutt. The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference*, pages 188–198, February 1957.

[4] John Backus. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformatiosn for high-performance computing. *ACM Computing Surveys*, *to appear.*

[6] John T. Bagwell, Jr. Local optimizations. *SIGPLAN Notices*, 5(7):52–66, July 1970. *Proceedings of a Symposium on Compiler Optimization.*

[7] J.C. Beatty. Register assignment algorithm for generation of highly optimized object code. *IBM Journal of Research and Development*, 1(18):20–39, January 1974.

[8] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, pages 78–101, 1966.

[9] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. *SIGPLAN Notices*, 32(6):287–295, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[10] David Bernstein, Dina Q. Goldin, Martin C. Golumbic, Hugo Krawczyk, Yishay Mansour, Itai Nahshon, and Ron Y. Pinter. Spill code minimization techniques for optimizing compilers. *SIGPLAN Notices*, 24(7):258–263, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[11] Robert L. Bernstein. Producing good code for the case statement. *Software—Practice and Experience*, 15(10):1021–1024, October 1985.

[12] Andrew Binstock and John Rex. *Practical Algorithms for Programmers.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[13] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, April 1992.

[14] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software–Practice and Experience*, 00(00), June 1997. Also available as a Technical Report From Center for Research on Parallel Computation, Rice University, number 95517-S.

[15] Preston Briggs, Keith D. Cooper, and Linda Torczon. Coloring register pairs. *ACM Letters on Programming Languages and Systems*, 1(1):3–13, March 1992.

[16] Preston Briggs, Keith D. Cooper, and Linda Torczon. Rematerialization. *SIGPLAN Notices*, 27(7):311–321, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[17] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[18] Michael Burke and Linda Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems*, 15(3):367–399, July 1993.

[19] David Callahan and Brian Koblenz. Register allocation via hierarchical graph coloring. *SIGPLAN Notices*, 26(6):192–203, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[20] Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction.*

[21] Gregory J. Chaitin. Register allocation and spilling via graph coloring. United States Patent 4,571,678, February 1986.

[22] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via graph coloring. *Computer Languages*, 6(1):47–57, January 1981.

[23] R.W. Conway and T.R. Wilcox. Design and implementation of a diagnostic compiler for PL/I. *Communications of the ACM*, pages 169–179, 1973.

[24] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

[25] Keith D. Cooper and L. Taylor Simpson. Optimistic global value numbering. *In preparation.*

[26] Keith D. Cooper and L. Taylor Simpson. Live range splitting in a graph coloring register allocator. In *Proceedings of the Seventh International Compiler Construction Conference, CC '98, Lecture Notes in Computer Science 1383*, 1998.

[27] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1992.

[28] Jack W. Davidson and Christopher W. Fraser. Register allocation and exhaustive peephole optimization. *Software–Practice and Experience*, 14(9):857–865, September 1984.

[29] Andrei P. Ershov. Reduction of the problem of memory allocation in programming to the problem of coloring the vertices of graphs. *Doklady Akademii Nauk S.S.S.R.*, 142(4), 1962. English translation in *Soviet Mathematics* 3:163–165, 1962.

[30] Andrei P. Ershov[2] Alpha – an automatic programming system of high efficiency. *Journal of the ACM*, 13(1):17–24, January 1966.

[31] R.W. Floyd. An algorithm for coding efficient arithmetic expressions. *Communications of the ACM*, pages 42–51, January 1961.

[32] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[33] Lal George and Andrew W. Appel. Iterated register coalescing. In *Conference Record of POPL '96: 23rd ACM SIGPLAN/SIGACT Symposium on Princples of Programming Languages*, pages 208–218, St. Petersburg, Florida, January 1996.

[2] The modern spelling is *Ershov.* Older variations include *Yershov* and *Eršov.*

[34] Rajiv Gupta, Mary Lou Soffa, and Tim Steele. Register allocation via clique separators. *SIGPLAN Notices*, 24(7):264–274, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation.*

[35] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, January 1990.

[36] William Harrison. A class of register allocation algorithms. Technical report, IBM Thomas J. Watson Research Center, 1975.

[37] Donald E. Knuth. *The Art Of Computer Programming.* Addison-Wesley, 1973.

[38] Donald E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Searching.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

[39] Steven M. Kurlander and Charles N. Fischer. Zero-cost range splitting. *SIGPLAN Notices*, 29(6):257–265, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[40] S. S. Lavrov. Store economy in closed operator schemes. *Journal of Computational Mathematics and Mathematical Physics*, 1(4):687–701, 1961. English translation in *U.S.S.R. Computational Mathematics and Mathematical Physics* 3:810-828, 1962.

[41] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[42] Vincenczo Liberatore, Martin Farach, and Ulrich Kremer. Hardness and algorithms for local register allocation. Technical Report LCSR-TR332, Rutgers University, June 1997.

[43] Todd A. Proebsting and Charles N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. *SIGPLAN Notices*, 26(6):256–267, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[44] Todd A. Proebsting and Charles N. Fischer. Probalistic register allocation. *SIGPLAN Notices*, 27(7):300–310, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[45] R.T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Proceedings of the Eastern Joint Computer Conference*, pages 133–138. Spartan Books, NY, USA, December 1959.

[46] Philip J. Schielke. *To be announced*. PhD thesis, Rice University, August 1999.

[47] Ravi Sethi and Jeffrey D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(7):715–728, July 1970.

[48] L. Taylor Simpson. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, 1996.

[49] Philip H. Sweany and Steven J. Beaty. Dominator-path scheduling—a global scheduling method. *SIGMICRO Newsletter*, 23(12):260–263, December 1992. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*.

[50] Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[51] William Wulf, Richard K. Johnson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke. *The Design of an Optimizing Compiler*. Programming Language Series. American Elsevier Publishing Company, 1975.