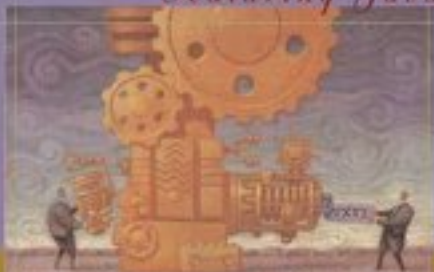


CRAFTING A COMPILER

Featuring Java



FISCHER • LeBLANC • CYTRON

Second Edition

1

Introduction

This chapter presents the basics of compiler history and organization. We begin in Section 1.1 with an overview of compilers and the history of their development. From there, we explain in Section 1.2 what a compiler does and how various compilers can be distinguished from each other: by the kind of machine code they generate and by the format of the target code they generate.

In Section 1.3, we discuss a kind of language processor called an *interpreter* and explain how an interpreter differs from a compiler. Section 1.4 discusses the *syntax* (structure) and *semantics* (meaning) of programs. Next, in Section 1.5, we discuss the tasks that a compiler must perform, primarily *analysis* of the source program and *synthesis* of a target program. That section also covers the parts of a compiler, discussing each in some detail: scanner, parser, type checker, optimizer and code generator.

In Section 1.6, we discuss the mutual interaction of compiler design and programming language design. Similarly, in Section 1.7 the influence of computer architecture on compiler design is covered.

Section 1.8 introduces a number of important compiler variants, including *debugging* and *development compilers*, *optimizing compilers*, and *retargetable compilers*. Finally, in Section 1.9, we consider *program development environments* that integrate a compiler, editor and debugger into a single tool.

1.1 Overview and History of Compilation

Compilers are fundamental to modern computing. They act as *translators*, transforming human-oriented *programming languages* into computer-oriented *machine languages*. To most users, a compiler can be viewed as a “black box” that performs the transformation illustrated in Figure 1.1. A compiler allows virtually all computer users to ignore the machine-dependent details of machine

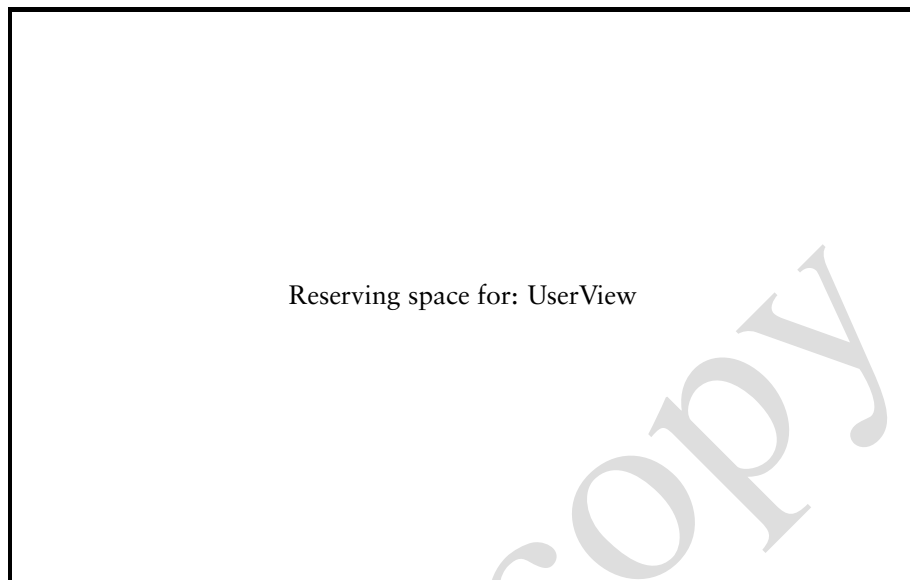


Figure 1.1: A user's view of a compiler.

language. Compilers therefore allow programs and programming expertise to be *machine-independent*. This is a particularly valuable capability in an age in which the number and variety of computers continue to grow explosively.

The term *compiler* was coined in the early 1950s by Grace Murray Hopper. Translation was then viewed as the “compilation” of a sequence of machine-language subprograms selected from a library. Compilation (as we now know it) was called “automatic programming,” and there was almost universal skepticism that it would ever be successful. Today, the automatic translation of programming languages is an accomplished fact, but programming language translators are still called “compilers.”

Among the first real compilers in the modern sense were the FORTRAN compilers of the late 1950s. They presented the user with a problem-oriented, largely machine-independent source language. They also performed some rather ambitious optimizations to produce efficient machine code, since efficient code was deemed essential for FORTRAN to compete successfully against then-dominant assembly languages. These FORTRAN compilers proved the viability of *high-level* (that is, mostly machine-independent) compiled languages and paved the way for the flood of languages and compilers that was to follow.

In the early days, compilers were *ad hoc* structures; components and techniques were often devised as a compiler was built. This approach to constructing compilers lent an aura of mystery to them, and they were viewed as complex and costly. Today the compilation process is well-understood and compiler construction is routine. Nonetheless, crafting an efficient and reliable compiler is still a complex task. Thus, in this book, we seek first to help you master the

fundamentals and then we explore some important innovations.

Compilers normally translate conventional programming languages like Java, C, and C++ into executable machine-language instructions. Compiler technology, however, is far more broadly applicable and has been employed in rather unexpected areas. For example, text-formatting languages like \TeX and \LaTeX [Lam95] are really compilers; they translate text and formatting commands into detailed typesetting commands. PostScript [Ado90], which is generated by text-formatters like \LaTeX , Microsoft Word, and Adobe FrameMaker, is really a programming language. It is translated and executed by laser printers and document previewers to produce a readable form of a document. This standardized document representation language allows documents to be freely interchanged, independent of how they were created and how they will be viewed.

Mathematica [Wol96] is an interactive system that intermixes programming with mathematics. With it, one can solve intricate problems in both symbolic and numeric forms. This system relies heavily on compiler techniques to handle the specification, internal representation, and solution of problems.

Languages like Verilog [Pal96] and VHDL [Coe89] address the creation of VLSI circuits. A **silicon compiler** specifies the layout and composition of a VLSI circuit mask, using standard cell designs. Just as an ordinary compiler must understand and enforce the rules of a particular machine language, a silicon compiler must understand and enforce the design rules that dictate the feasibility of a given circuit.

Compiler technology, in fact, is of value in almost any program that presents a nontrivial text-oriented command set, including the command and scripting languages of operating systems and the query languages of database systems. Thus, while our discussion will focus on traditional compilation tasks, innovative readers will undoubtedly find new and unexpected applications for the techniques presented.

1.2 What Compilers Do

Figure 1.1 represents a compiler as a translator of the programming language being compiled (the *source*) to some machine language (the *target*). This description suggests that all compilers do about the same thing, the only difference being their choice of source and target languages. In fact, the situation is a bit more complicated. While the issue of the accepted source language is indeed simple, there are many alternatives in describing the output of a compiler. These go beyond simply naming a particular target computer. Compilers may be distinguished in two ways:

- By the kind of machine code they generate
- By the format of the target code they generate

These are discussed in the following sections.

1.2.1 Distinguishing Compilers by the Machine Code Generated

Compilers may generate any of three types of code by which they can be differentiated:

- Pure Machine Code
- Augmented Machine Code
- Virtual Machine Code

Pure Machine Code

Compilers may generate code for a particular machine's instruction set, not assuming the existence of any operating system or library routines. Such machine code is often called **pure code** because it includes nothing but instructions that are part of that instruction set. This approach is rare. It is most commonly used in compilers for **system implementation languages**—languages intended for implementing operating systems or embedded applications (like a programmable controller). This form of target code can execute on bare hardware without dependence on any other software.

Augmented Machine Code

Far more often, compilers generate code for a machine architecture that is **augmented** with operating system routines and run-time language support routines. The execution of a program generated by such a compiler requires that a particular operating system be present on the target machine and a collection of language-specific run-time support routines (I/O, storage allocation, mathematical functions, and so on) be available to the program. The combination of the target machine's instruction set and these operating system and language support routines can be thought of as defining a **virtual machine**. A virtual machine is a machine that exists only as a hardware/software combination.

The degree to which the virtual machine matches the actual hardware can vary greatly. Some common compilers translate almost entirely to hardware instructions; for example, most FORTRAN compilers use software support only for I/O and mathematical functions. Other compilers assume a wide range of virtual instructions. These may include data transfer instructions (for example, to move bit fields), procedure call instructions (to pass parameters, save registers, allocate stack space, and so on), and dynamic storage instructions (to provide for heap allocation).

Virtual Machine Code

The third type of code generated is composed entirely of virtual instructions. This approach is particularly attractive as a technique for producing a **transportable compiler**, a compiler that can be run easily on a variety of computers. Transportability is enhanced because moving the compiler entails only writing a

simulator for the virtual machine used by the compiler. This applies, however, only if the compiler **bootstraps**—compiles itself—or is written in an available language. If this virtual machine is kept simple and clean, the interpreter can be quite easy to write. Examples of this approach are early Pascal compilers and the Java compiler included in the Java Development Kit [Sun98]. Pascal uses P-code [Han85], while Java uses **Java virtual machine (JVM)** code. Both represent virtual stack machines. A decent simulator for P-code or JVM code can easily be written in a few weeks. Execution speed is roughly five to ten times slower than that of compiled code. Alternatively, the virtual machine code can be either translated into C code or expanded to machine code directly. This approach made Pascal and Java available for almost any platform. It was instrumental in Pascal's success in the 1970s and has been an important factor in Java's growing popularity.

As can be seen from the preceding discussion, virtual instructions serve a variety of purposes. They simplify the job of a compiler by providing primitives suitable for the particular language being translated (such as procedure calls and string manipulation). They also contribute to compiler transportability. Further, they may allow for a significant *decrease* in the size of generated code—instructions can be designed to meet the needs of a particular programming language (for example, JVM code for Java). Using this approach, one can realize as much as a two-thirds reduction in generated program size. When a program is transmitted over a slow communications path (e.g., a Java applet sent from a slow server), size can be a crucial factor.

When an entirely virtual instruction set is used as the target language, the instruction set must be interpreted (simulated) in software. In a **just-in-time (JIT)** approach, virtual instructions can be translated to target code just as they are about to be executed or when they have been interpreted often enough to merit translation into target code.

If a virtual instruction set is used often enough, it is even possible to develop special microprocessors (such as the PicoJava processor by Sun Microsystems) that directly implement the virtual instruction set in hardware.

To summarize, almost all compilers, to a greater or lesser extent, generate code for a virtual machine, some of whose operations must be interpreted in software or firmware. We consider them compilers because they make use of a distinct translation phase that precedes execution.

1.2.2 Target Code Formats

Another way that compilers differ from one another is in the format of the target code they generate. Target formats may be categorized as follows:

- Assembly language
- Relocatable binary
- Memory-image

Assembly Language (Symbolic) Format

The generation of assembly code simplifies and modularizes translation. A number of code generation decisions (how to compute addresses, whether to use absolute or relative addressing, and so on) can be left for the assembler. This approach is common among compilers that were developed either as instructional projects or to support experimental programming language designs.

Generating assembler code is also useful for **cross-compilation** (running a compiler on one computer, with its target language being the machine language of a second computer). This is because a symbolic form is produced that is easily transferred between different computers. This approach also makes it easier to check the correctness of a compiler, since its output can be observed.

Often C, rather than a specific assembly language, is generated, with C's being used as a “universal assembly language.” C is far more platform-independent than any particular assembly language. However, some aspects of a program (such as the run-time representations of program and data) are inaccessible using C code, while they are readily accessible in assembly language.

Most production-quality compilers do not generate assembly language; direct generation of target code (in relocatable or binary format, discussed next) is more efficient. However, the compiler writer still needs a way to check the correctness of generated code. Thus it is wise to design a compiler so that it optionally will produce **pseudoassembly language**, that is, a listing of what the assembly language would look like if it were produced.

Relocatable Binary Format

Target code also may be generated in a **binary format**. In this case, external references and local instruction and data addresses are not yet bound. Instead, addresses are assigned relative either to the beginning of the module or to symbolically named locations. (This latter alternative makes it easy to group together code sequences or data areas.) This format is the typical output of an assembler, so this approach simply eliminates one step in the process of preparing a program for execution. A linkage step is required to add any support libraries and other separately compiled routines referenced from within a compiled program and to produce an **absolute binary program** format that is executable.

Both relocatable binary and assembly language formats allow **modular compilation**, the breaking up of a large program into separately compiled pieces. They also allow **cross-language references**, calls of assembler routines or subprograms in other high-level languages. Further, they support subroutine libraries, for example, I/O, storage allocation, and math routines.

Memory-Image (Absolute Binary) Form

The compiled output may be loaded into the compiler's address space and immediately executed, instead of being left in a file as with the first two approaches. This process is usually much faster than going through the intermediate step of

link/editing. It also allows a program to be prepared and executed in a single step. However, the ability to interface with external, library, and precompiled routines may be limited. Further, the program must be recompiled for each execution unless some means is provided for storing the memory image. Memory-image compilers are useful for student and debugging use, where frequent changes are the rule and compilation costs far exceed execution costs. It also can be useful to *not* save absolutes after compilation (for example, in order to save file space or to guarantee the use of only the most current library routines and class definitions).

Java is designed to use and share classes defined and implemented at a variety of organizations. Rather than use a fixed copy of a class (which may be outdated), the JVM supports **dynamic linking** of externally defined classes. That is, when a class is first referenced, a class definition may be remotely fetched, checked, and loaded during program execution. In this way, “foreign code” can be guaranteed to be up-to-date and secure.

The code format alternatives and the target code alternatives discussed here show that compilers can differ quite substantially while still performing the same sort of translation task.

1.3 Interpreters

Another kind of language processor is the **interpreter**. An interpreter differs from a compiler in that it executes programs without explicitly performing a translation. Figure 1.2 illustrates schematically how interpreters work.

Interpreters behave differently than a compiler. To an interpreter, a program is merely data that can be arbitrarily manipulated, just like any other data. The locus of control during execution resides in the interpreter, *not* in the user program (that is, the user program is passive rather than active).

Interpreters provide a number of capabilities not found in compilers, as follows.

- Programs may be modified as execution proceeds. This provides a straightforward interactive debugging capability. Depending on program structure, program modifications may require reparsing or repeated semantic analysis.
- Languages in which the type of object a variable denotes may change dynamically (e.g., Lisp and Scheme) are easily supported in an interpreter. Since the user program is continuously reexamined as execution proceeds, symbols need not have a fixed meaning (for example, a symbol may denote an integer scalar at one point and a Boolean array at a later point). Such **fluid bindings** are obviously much more troublesome for compilers, since dynamic changes in the meaning of a symbol make direct translation into machine code impossible.

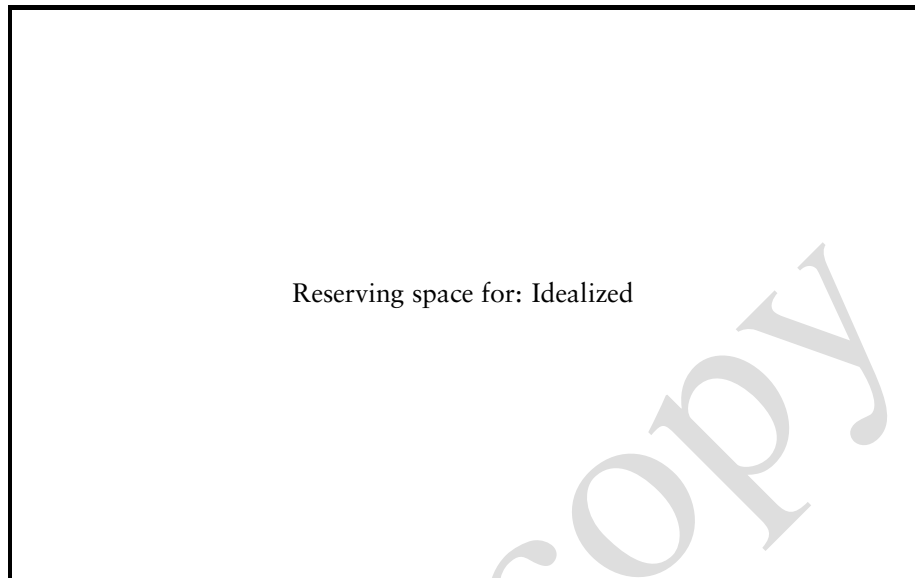


Figure 1.2: An idealized interpreter.

- Interpreters can provide better diagnostics. Since source text analysis is intermixed with program execution, especially good diagnostics (re-creation of source lines in error, use of variable names in error messages, and so on) are produced more easily than they are by compilers.
- Interpreters provide a significant degree of machine independence, since no machine code is generated. All operations are performed within the interpreter. To move an interpreter to a new machine, you need only recompile the interpreter on the new machine.

However, direct interpretation of source programs can involve large overheads, as follows.

- As execution proceeds, program text must be continuously reexamined, with identifier bindings, types, and operations sometimes recomputed at each reference. For very dynamic languages, this can represent a 100 to 1 (or worse) factor in execution speed over compiled code. For more static languages (such as C or Java), the speed degradation is closer to a factor of 5 or 10 to 1.
- Substantial space overhead may be involved. The interpreter and all support routines must usually be kept available. Source text is often not as compact as if it were compiled (for example, symbol tables are present and program text may be stored in a format designed for easy access and modification rather than for space minimization). This size penalty may

lead to restrictions in, for example, the size of programs and the number of variables or procedures. Programs beyond these built-in limits cannot be handled by the interpreter.

- Many languages (for example, C, C++, and Java) have both interpreters (for debugging and program development) and compilers (for production work).

In summary, all language processing involves interpretation at some level. Interpreters directly interpret source programs or some syntactically transformed versions of them. They may exploit the availability of a source representation to allow program text to be changed as it is executed and debugged. While a compiler has distinct translation and execution phases, some form of “interpretation” is still involved. The translation phase may generate a virtual machine language that is interpreted by software or a real machine language that is interpreted by a particular computer, either in firmware or hardware.

1.4 Syntax and Semantics of Programming Languages

A complete definition of a programming language must include the specification of its *syntax* (structure) and its *semantics* (meaning).

Syntax typically means context-free syntax because of the almost universal use of **context-free grammars** (CFGs) as a syntactic specification mechanism. Syntax defines the sequences of symbols that are legal; syntactic legality is independent of any notion of what the symbols mean. For example, a context-free syntax might say that $a = b + c$ is syntactically legal, while $b + c = a$ is not. Not all program structure can be described by context-free syntax, however. For example, CFGs cannot specify type compatibility and scoping rules (for instance, that $a = b + c$ is illegal if any of the variables is undeclared or if b or c is of type Boolean).

Because of the limitations of CFGs, the semantic component of a programming language is commonly divided into two classes:

- Static semantics
- Run-time semantics

1.4.1 Static Semantics

The **static semantics** of a language is a set of restrictions that determine which syntactically legal programs are actually valid. Typical static semantic rules require that all identifiers be declared, that operators and operands be type-compatible, and that procedures be called with the proper number of parameters. The common thread through all of these rules is that they cannot be expressed with a CFG. Static semantics thus *augment* context-free specifications and complete the definition of valid programs.

Static semantics can be specified formally or informally. The prose feature descriptions found in most programming language manuals are informal specifications. They tend to be relatively compact and easy to read, but often they are imprecise. Formal specifications might be expressed using any of a variety of notations. For example, **attribute grammars** [Knu68] are a popular method of formally specifying static semantics. They formalize the semantic checks commonly found in compilers. The following rewriting rule, called a *production*, specifies that an expression, denoted by E, can be rewritten into an expression, E, plus a term, T.

$$E \rightarrow E + T$$

In an attribute grammar, this production might be augmented with a type attribute for E and T and a predicate testing for type compatibility, such as

$$E_{result} \rightarrow E_{v1} + T_{v2}$$

```

    if v1.type = numeric and v2.type = numeric
    then result.type ← numeric
    else call ERROR( )
  
```

Attribute grammars are a reasonable blend of formality and readability, but they can still be rather verbose. A number of compiler writing systems employ attribute grammars and provide automatic evaluation of attribute values [RT88].

1.4.2 Run-time Semantics

Run-time, or **execution semantics** are used to specify what a program computes. These semantics are often specified very informally in a language manual or report. Alternatively, a more formal *operational*, or *interpreter*, model can be used. In such a model, a program “state” is defined and program execution is described in terms of changes to that state. For example, the semantics of the statement $a = 1$ is that the state component corresponding to a is changed to 1.

A variety of formal approaches to defining the run-time semantics of programming languages have been developed. Three of them, natural semantics, axiomatic semantics and denotational semantics, are described below.

Natural Semantics

Natural semantics [NN92] (sometimes called **structured operational semantics**) formalizes the operational approach. Given assertions known to be true before the evaluations of a construct, we can infer assertions that will hold after the construct's evaluation. Natural semantics has been used to define the semantics of a variety of languages, including standard ML [MTH90].

Axiomatic Definitions

Axiomatic definitions [Gri81] can be used to model execution at a more abstract level than operational models. They are based on formally specified *relations*, or *predicates*, that relate program variables. Statements are defined by how they modify these relations.

As an example of axiomatic definitions, the axiom defining $var \leftarrow exp$, states that a predicate involving var is **true** after statement execution if, and only if, the predicate obtained by replacing all occurrences of var by exp is **true** beforehand. Thus, for $y > 3$ to be **true** after execution of the statement $y \leftarrow x + 1$, the predicate $x + 1 > 3$ would have to be **true** before the statement is executed. Similarly, $y = 21$ is **true** after execution of $x \leftarrow 1$ if $y = 21$ is **true** before its execution—this is a roundabout way of saying that changing x doesn't affect y . However, if x is an **alias** (an alternative name) for y , the axiom is invalid. This is one reason why aliasing is discouraged (or forbidden) in modern language designs.

The axiomatic approach is good for deriving proofs of program correctness because it avoids implementation details and concentrates on how relations among variables are changed by statement execution. Although axioms can formalize important properties of the semantics of a programming language, it is difficult to use them to define most programming languages completely. For example, they do not do a good job of modeling implementation considerations such as running out of memory.

Denotational Models

Denotational models [Sch86] are more mathematical in form than operational models. Yet they still present notions of memory access and update that are central to procedural languages. They rely on notation and terminology drawn from mathematics, so they are often fairly compact, especially in comparison with operational definitions.

A denotational definition may be viewed as a syntax-directed definition that specifies the meaning of a construct in terms of the meaning of its immediate constituents. For example, to define addition, we might use the following rule:

$$E[T1 + T2]m = E[T1]m + E[T2]m$$

This definition says that the value obtained by adding two subexpressions, $T1$ and $T2$, in the context of a memory state m is defined to be the sum of the arithmetic values obtained by evaluating $T1$ in the context of m (denoted $E[T1]m$) and $T2$ in the context of m (denoted $E[T2]m$).

Denotational techniques are quite popular and form the basis for rigorous definitions of programming languages. Research has shown it is possible to convert denotational representations *automatically* to equivalent representations that are directly executable [Set83, Wan82, App85].

Again, our concern for precise semantic specification is motivated by the fact that writing a complete and accurate compiler for a programming language requires that the language itself be well-defined. While this assertion may seem

self-evident, many languages are defined by imprecise language reference manuals. Such a manual typically includes a formal syntax specification but otherwise is written in an informal prose style. The resulting definition inevitably is ambiguous or incomplete on certain points. For example, in Java all functions must return via a `return expr` statement, where `expr` is assignable to the function's return type. Thus

```
public static int subr(int b) {
    if (b != 0)
        return b+100;
}
```

is illegal, since if `b` is equal to zero, `subr` will not return a value. But what about this:

```
public static int subr(int b) {
    if (b != 0)
        return b+100;
    else if (10*b == 0)
        return 1;
}
```

In this case, a proper return is always executed, since the `else` part is reached only if `b` equals zero; this implies that `10*b` is also equal to zero. Is the compiler expected to duplicate this rather involved chain of reasoning? Although the Java reference manual doesn't explicitly say so, there is an implicit "all paths reachable" assumption that allows a compiler to assume that both legs of a conditional are executable even if a detailed program analysis shows this to be untrue. Thus a compiler may reject `subr` as semantically illegal and in so doing trade simplicity for accuracy in its analysis. Indeed, the general problem of deciding whether a particular statement in a program is reachable is *undecidable* (this is a variant of the famous *halting problem* [HU79]). We certainly can't ask our Java compiler literally to do the impossible!

In practice, a trusted **reference compiler** can serve as a *de facto* language definition. That is, a programming language is, in effect, defined by what a compiler chooses to accept and how it chooses to translate language constructs. In fact, the operational and natural semantic approaches introduced previously take this view. A standard interpreter is defined for a language, and the meaning of a program is precisely whatever the interpreter says. An early (and very elegant) example of an operational definition is the seminal Lisp interpreter [McC65]. There, all of Lisp was defined in terms of the actions of a Lisp interpreter, assuming only seven primitive functions and the notions of argument binding and function call.

Of course, a reference compiler or interpreter is no substitute for a clear and precise semantic definition. Nonetheless, it is very useful to have a reference against which to test a compiler that is under development.

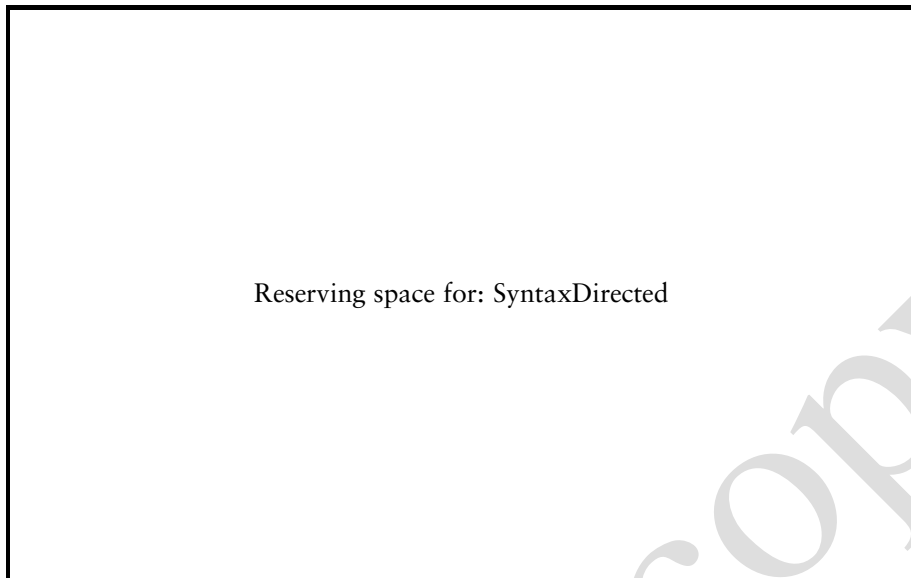


Figure 1.3: A syntax-directed compiler.

1.5 Organization of a Compiler

Any compiler must perform two major tasks:

1. *Analysis* of the source program being compiled
2. *Synthesis* of a target program that, when executed, will correctly perform the computations described by the source program

Almost all modern compilers are **syntax-directed**. That is, the compilation process is driven by the syntactic structure of the source program, as recognized by the parser. The parser builds this structure out of **tokens**, the elementary symbols used to define a programming language syntax. Recognition of syntactic structure is a major part of the **syntax analysis** task.

Semantic analysis examines the meaning (semantics) of the program on the basis of its syntactic structure. It plays a dual role. It finishes the analysis task by performing a variety of correctness checks (for example, enforcing type and scope rules). It also begins the *synthesis phase*.

In the synthesis phase, either source language constructs are translated into some **intermediate representation** (IR) of the program or target code may be directly generated. If an IR is generated, it then serves as input to a *code generator* component that actually produces the desired machine-language program. The IR may optionally be transformed by an *optimizer* so that a more efficient program may be generated. A common organization of all of these compiler

components is depicted schematically in Figure 1.3. The following subsections describe the components of a compiler:

- Scanner
- Parser
- Type checker
- Optimizer
- Code generator

Chapter Chapter:global:two presents a simple compiler to provide concrete examples of many of the concepts introduced in this overview.

1.5.1 The Scanner

The **scanner** begins the analysis of the source program by reading the input text—character by character—and grouping individual characters into tokens—identifiers, integers, reserved words, delimiters, and so on. This is the first of several steps that produce successively higher-level representations of the input. The tokens are encoded (often as integers) and are fed to the parser for syntactic analysis. When necessary, the actual character string comprising the token is also passed along for use by the semantic phases. The scanner does the following.

- It puts the program into a compact and uniform format (a stream of tokens).
- It eliminates unneeded information (such as comments).
- It processes compiler control directives (for example, turn the listing on or off and include source text from a file).
- It sometimes enters preliminary information into symbol tables (for example, to register the presence of a particular label or identifier).
- It optionally formats and lists the source program.

The main action of building tokens is often driven by token descriptions. *Regular expression* notation (discussed in Chapter Chapter:global:three) is an effective approach to describing tokens. Regular expressions are a formal notation sufficiently powerful to describe the variety of tokens required by modern programming languages. In addition, they can be used as a specification for the automatic generation of finite automata (discussed in Chapter Chapter:global:three) that recognize **regular sets**, that is, the sets that regular expressions define. Recognition of regular sets is the basis of the **scanner generator**. A scanner generator is a program that actually produces a working scanner when given only a specification of the tokens it is to recognize. Scanner generators are a valuable compiler-building tool.

1.5.2 The Parser

The **parser**, when given a formal syntax specification, typically as a CFG, reads tokens and groups them into phrases as specified by the *productions* of the CFG being used. (Grammars are discussed in Chapters Chapter:global:two and Chapter:global:four; parsing is discussed in Chapters Chapter:global:five and Chapter:global:six.) Parsers are typically driven by tables created from the CFG by a *parser generator*.

The parser verifies correct syntax. If a syntax error is found, it issues a suitable error message. Also, it may be able to repair the error (to form a syntactically valid program) or to recover from the error (to allow parsing to be resumed). In many cases, syntactic error recovery or repair can be done automatically by consulting *error-repair* tables created by a parser/repair generator.

As syntactic structure is recognized, the parser usually builds an **abstract syntax tree** (AST). An abstract syntax tree is a concise representation of program structure, that is used to guide semantic processing. Abstract syntax trees are discussed in Chapter Chapter:global:two, seven.

1.5.3 The Type Checker (Semantic Analysis)

The type checker checks the **static semantics** of each AST node. That is, it verifies that the construct the node represents is legal and meaningful (that all identifiers involved are declared, that types are correct, and so on). If the construct is semantically correct, the type checker “decorates” the AST node by adding type information to it. If a semantic error is discovered, a suitable error message is issued.

Type checking is purely dependent on the semantic rules of the source language. It is independent of the compiler's target.

Translator (Program Synthesis)

If an AST node is semantically correct, it can be *translated*. That is, IR code that correctly implements the construct the AST represents is generated. Translation involves capturing the run-time meaning of a construct.

For example, an AST for a while loop contains two subtrees, one representing the loop's expression and the other representing the loop's body. *Nothing* in the AST captures the notion that a while loop loops! This meaning is captured when a while loop's AST is translated to IR form. In the IR, the notion of testing the value of the loop control expression and conditionally executing the loop body is made explicit.

The translator is largely dictated by the semantics of the source language. Little of the nature of the target machine needs to be made evident. As a convenience during translation, some general aspects of the target machine may be exploited (for example, that the machine is byte-addressable and that it has a run-time stack). However, detailed information on the nature of the target machine

(operations available, addressing, register characteristics, and so on) is reserved for the code-generation phase.

In simple, nonoptimizing compilers, the translator may generate target code directly without using an explicit intermediate representation. This simplifies a compiler's design by removing an entire phase. However, it also makes re-targeting the compiler to another machine much more difficult. Most compilers implemented as instructional projects generate target code directly from the AST, without using an IR.

More elaborate compilers may first generate a high-level IR (that is source language-oriented) and then subsequently translate it into a low-level IR (that is target machine-oriented). This approach allows a cleaner separation of source and target dependencies.

Symbol Tables

A **symbol table** is a mechanism that allows information to be associated with identifiers and shared among compiler phases. Each time an identifier is declared or used, a symbol table provides access to the information collected about it. Symbol tables are used extensively during type checking, but they can also be used by other compiler phases to enter, share, and later retrieve information about variables, procedures, labels, and so on. Compilers may choose to use other structures to share information between compiler phases. For example, a program representation such as an AST may be expanded and refined to provide detailed information needed by optimizers, code generators, linkers, loaders, and debuggers.

1.5.4 The Optimizer

The IR code generated by the translator is analyzed and transformed into functionally equivalent but improved IR code by the **optimizer**. This phase can be complex, often involving numerous subphases, some of which may need to be applied more than once. Most compilers allow optimizations to be turned off so as to speed translation. Nonetheless, a carefully designed optimizer can significantly speed program execution by simplifying, moving, or eliminating unneeded computations.

If both a high-level and low-level IR are used, optimizations may be performed in stages. For example, a simple subroutine call may be expanded into the subroutine's body, with actual parameters substituted for formal parameters. This is a high-level optimization. Alternatively, a value already loaded from memory may be reused. This is a low-level optimization.

Optimization can also be done *after* code generation. An example is **peephole optimization**. Peephole optimization examines generated code a few instructions at a time (in effect, through a “peephole”). Common peephole optimizations include eliminating multiplications by one or additions of zero, eliminating a load of a value into a register when the value is already in another register, and replacing a sequence of instructions by a single instruction with the same effect.

A peephole optimizer does not offer the payoff of a full-scale optimizer. However, it can significantly improve code and is often useful for “cleaning up” after earlier compiler phases.

1.5.5 The Code Generator

The IR code produced by the translator is mapped into target machine code by the **code generator**. This phase requires detailed information about the target machine and includes machine-specific optimization such as register allocation and code scheduling. Normally, code generators are hand-coded and can be quite complex, since generation of good target code requires consideration of many special cases.

The notion of *automatic construction* of code generators has been actively studied. The basic approach is to match a low-level IR to target-instruction templates, with the code generator automatically choosing instructions that best match IR instructions. This approach localizes the target-machine dependences of a compiler and, at least in principle, makes it easy to **retarget** a compiler to a new target machine. Automatic retargeting is an especially desirable goal, since a great deal of work is usually needed to move a compiler to a new machine. The ability to retarget by simply changing the set of target machine templates and generating (from the templates) a new code generator is compelling.

A well-known compiler using these techniques is the GNU C compiler [Sta89], `gcc`. `gcc` is a heavily optimizing compiler that has machine description files for more than ten popular computer architectures and at least two language front ends (C and C++).

1.5.6 Compiler Writing Tools

Finally, note that in discussing compiler design and construction, we often talk of *compiler writing tools*. These are often packaged as *compiler generators* or *compiler-compilers*. Such packages usually include scanner and parser generators. Some also include symbol table routines, attribute grammar evaluators, and code-generation tools. More advanced packages may aid in error repair generation.

These sorts of generators greatly aid in building pieces of compilers, but much of the effort in building a compiler lies in writing and debugging the semantic phases. These routines are numerous (a type checker and translator are needed for each distinct AST node) and are usually hand-coded.

1.6 Compiler Design and Programming Language Design

Our primary interest is the design and implementation of compilers for modern programming languages. An interesting aspect of this study is how programming language design and compiler design influence one another. Obviously,

programming language design influences, and indeed often dictates, compiler design. Many clever and sometimes subtle compiler techniques arise from the need to cope with some programming language construct. A good example of this is the **closure** mechanism that was invented to handle formal procedures. A closure is a special run-time representation for a function. It consists of a pointer to the function's body *and* to its execution environment.

The state of the art in compiler design also strongly affects programming language design, if only because a programming language that cannot be compiled effectively will usually remain unused! Most successful programming language designers (such as the Java development team) have extensive compiler design backgrounds.

A programming language that is easy to compile has many advantages, as follows.

- It often is easier to learn, read, and understand. If a feature is hard to compile, it may well be hard to understand.
- It will have quality compilers on a wide variety of machines. This fact is often crucial to a language's success. For example, C, C++, and FORTRAN are widely available and very popular; Ada and Modula-3 have limited availability and are far less popular.
- Often better code will be generated. Poor quality code can be fatal in major applications.
- Fewer compiler bugs will occur. If a language can't be understood, how can it be effectively compiled?
- The compiler will be smaller, cheaper, faster, more reliable, and more widely used.
- Compiler diagnostics and program development tools will often be better.

Throughout our discussion of compiler design, we draw ideas, solutions, and shortcomings from many languages. Our primary focus is on Java and C, but we also consider Ada, C++, SmallTalk, ML, Pascal, and FORTRAN. We concentrate on Java and C because they are representative of the issues posed by modern language designs. We consider other languages so as to identify alternative design approaches that present new challenges to compilation.

1.7 Architectural Influences of Computer Design

Advances in computer architecture and microprocessor fabrication have spearheaded the computer revolution. At one time, a computer offering one megaflop performance (1,000,000 floating-point operations per second) was considered advanced. Now computers offering in excess of 10 *teraflops* (10,000,000,000,000 floating-point operations per second) are under development.

Compiler designers are responsible for making this vast computing capability available to programmers. Although compilers are rarely visibly to the end users of application programs, they are an essential “enabling technology.” The problems encountered in efficiently harnessing the capability of a modern microprocessor are numerous, as follows.

- Instruction sets for some popular microprocessors, particularly the x86 series, are highly nonuniform. Some operations must be done in registers, while others can be done in memory. Often a number of register classes exist, each suitable for only a particular class of operations.
- High-level programming language operations are not always easy to support. Heap operations can take hundreds or thousands of machine instructions to implement. Exceptions and program threads are far more expensive and complex to implement than most users suspect.
- Essential architectural features such as hardware caches and distributed processors and memory are difficult to present to programmers in an architecturally independent manner. Yet misuse of these features can impose immense performance penalties.

Data and program integrity have been undervalued, and speed has been overemphasized. As a result, programming errors can go undetected because of a fear that extra checking will slow down execution unacceptably. A major complexity in implementing Java is efficiently enforcing the run-time integrity constraints it imposes.

1.8 Compiler Variants

Compilers come in many forms, including the following:

- Debugging, or development compilers
- Optimizing compilers
- Retargetable compilers

These are discussed in the following sections.

1.8.1 Debugging (Development) Compilers

A **debugging**, or **development** compiler such as `CodeCenter` [KLP88] or `Borland C++` [Sch97] is specially designed to aid in the development and debugging of programs. It carefully scrutinizes programs and details programmer errors. It also often can tolerate or repair minor errors (for example, insert a missing comma or parenthesis). Some program errors can be detected only at run-time. Such errors include invalid subscripts, misuse of pointers, and illegal file manipulations. A debugging compiler may include the checking of code that can detect

run-time errors and initiate a symbolic debugger. Although debugging compilers are particularly useful in instructional environments, diagnostic techniques are of value in all compilers. In the past, development compilers were used only in the initial stages of program development. When a program neared completion, a “production compiler,” which increased compilation and execution speed by ignoring diagnostic concerns, was used. This strategy has been likened by Tony Hoare to wearing a life jacket in sailing classes held on dry land but abandoning the jacket when at sea! Indeed, it is becoming increasingly clear that for almost all programs, correctness rather than speed is the paramount concern. Java, for example, mandates run-time checks that C and C++ ignore.

Ways of detecting and characterizing errors in heavily used application programs are of great interest. Tools such as `purify` [HJ92] can add initialization and array bounds checks to already compiled programs, thereby allowing illegal operations to be detected even when source files are not available.

1.8.2 Optimizing Compilers

An **optimizing compiler** is specially designed to produce efficient target code at the cost of increased compiler complexity and possibly increased compilation times. In practice, all production quality compilers—those whose output will be used in everyday work—make some effort to generate good target code. For example, no add instruction would normally be generated for the expression `i+0`.

The term *optimizing compiler* is actually a misnomer. This is because no compiler, no matter how sophisticated, can produce *optimal* code for all programs. The reason for this is twofold. First, theoretical computer science has shown that even so simple a question as whether two programs are equivalent is **undecidable**, that is, it cannot be solved by *any* computer program. Thus finding the simplest (and most efficient) translation of a program can't always be done. Second, many program optimizations require time proportional to an exponential function of the size of the program being compiled. Thus optimal code, even when theoretically possible, often is infeasible in practice.

Optimizing compilers actually use a wide variety of transformations that improve a program's performance. The complexity of an optimizing compiler arises from the need to employ a variety of transforms, some of which interfere with each other. For example, keeping frequently used variables in registers reduces their access time but makes procedure and function calls more expensive. This is because registers need to be saved across calls. Many optimizing compilers provide a number of levels of optimization, each providing increasingly greater code improvements at increasingly greater costs. The choice of which improvements are most effective (and least expensive) is a matter of judgment and experience. In later chapters, we suggest possible optimizations, with the emphasis on those that are both simple and effective. Discussion of a comprehensive optimizing compiler is beyond the scope of this book. However, compilers that produce high-quality code at reasonable cost are an achievable goal.

1.8.3 Retargetable Compilers

Compilers are designed for a particular programming language (the source language) and a particular target computer (the computer for which it will generate code). Because of the wide variety of programming languages and computers that exist, a large number of similar, but not identical, compilers must be written. While this situation has decided benefits for those of us in the compiler writing business, it does make for a lot of duplication of effort and for a wide variance in compiler quality. As a result, a new kind of compiler, the **retargetable compiler**, has become important.

A retargetable compiler is one whose target machine can be changed without its machine-independent components having to be rewritten. A retargetable compiler is more difficult to write than an ordinary compiler because target machine dependencies must be carefully localized. It also is often difficult for a retargetable compiler to generate code that is as efficient as that of an ordinary compiler because special cases and machine idiosyncrasies are harder to exploit. Nonetheless, because a retargetable compiler allows development costs to be shared and provides for uniformity across computers, it is an important innovation. While discussing the fundamentals of compilation, we concentrate on compilers targeted to a single machine. In later chapters, the techniques needed to provide retargetability will be considered.

1.9 Program Development Environment

In practice, a compiler is but one tool used in the edit-compile-test cycle. That is, a user first edits a program, then compiles it, and finally tests its performance. Since program bugs will inevitably be discovered and corrected, this cycle is repeated many times. A popular programming tool, the **program development environment** (PDE), has been designed to integrate this cycle within a single tool. A PDE allows programs to be built incrementally, with program checking and testing fully integrated. PDEs may be viewed as the next stage in the evolution of compilers.

We focus on the traditional **batch compilation** approach in which an entire source file is translated. Many of the techniques we develop can be reformulated into **incremental** form to support PDEs. Thus a parser can reparse only portions of a program that have been changed [WG97], and a type checker can analyze only portions of an abstract syntax tree that have been changed.

In this book, we concentrate on the translation of C, C++, and Java. We use the JVM as our target, but we also address popular microprocessor architectures, particularly RISC processors such as the MIPS [KHH91] and Sparc [WG94]. At the code-generation stage, a variety of current techniques designed to exploit fully a processor's capabilities will be explored. Like so much else in compiler design, experience is the best guide, so we start with the translation of a very simple language and work our way up to ever more challenging translation tasks.

Exercises

1. The model of compilation we introduced is essentially batch-oriented. In particular, it assumes that an entire source program has been written and that the program will be fully compiled before the programmer can execute the program or make any changes. An interesting and important alternative is an **interactive compiler**. An interactive compiler, usually part of an integrated program development environment, allows a programmer to interactively create and modify a program, fixing errors as they are detected. It also allows a program to be tested before it is fully written, thereby providing for stepwise implementation and testing.

Redesign the compiler structure of Figure 1.3 to allow incremental compilation. (The key idea is to allow individual phases of a compiler to be run or rerun without necessarily doing a full compilation.)

2. Most programming languages, such as C and C++, are compiled directly into the machine language of a “real” microprocessor (for example, an Intel x86 or DEC alpha). Java takes a different approach. It is commonly compiled into the machine language of the **Java virtual machine (JVM)**. The JVM isn't implemented in its own microprocessor but rather is interpreted on some existing processor. This allows Java to be run on a wide variety of machines, thereby making it highly “platform-independent.”

Explain why building an interpreter for a virtual machine like the JVM is easier and faster than building a complete Java compiler. What are the disadvantages of this virtual machine approach to compiler implementation?

3. C compilers are almost always written in C. This raises something of a “chicken and egg” problem—how was the *first* C compiler for a particular system created? If you need to create the first compiler for language X on system Y, one approach is to create a **cross-compiler**. A cross-compiler runs on system Z but generates code for system Y.

Explain how, starting with a compiler for language X that runs on system Z, you might use cross-compilation to create a compiler for language X, written in X, that runs on system Y and generates code for system Y.

What extra problems arise if system Y is “bare”—that is, has no operating system or compilers for any language? (Recall that Unix is written in C and thus must be compiled before its facilities can be used.)

4. Cross-compilation assumes that a compiler for language X exists on some machine. When the first compiler for a new language is created, this assumption doesn't hold. In this situation, a **bootstrapping** approach can be taken. First, a subset of language X is chosen that is sufficient to implement a simple compiler. Next, a simple compiler for the X subset is written in any available language. This compiler must be correct, but it should not be any more elaborate than is necessary, since it will soon be discarded. Next, the subset compiler for X is rewritten in the X subset and then compiled

using the subset compiler previously created. Finally, the X subset, and its compiler, can be enhanced until a complete compiler for X , written in X , is available.

Assume you are bootstrapping C++ or Java (or some comparable language). Outline a suitable subset language. What language features must be in the language? What other features are desirable?

5. To allow the creation of camera-ready documents, languages like $\text{T}_{\text{E}}\text{X}$ and $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ have been created. These languages can be thought of as varieties of programming language whose output controls laser printers or phototypesetters. Source language commands control details like spacing, font choice, point size, and special symbols. Using the syntax-directed compiler structure of Figure 1.3, suggest the kind of processing that might occur in each compiler phase if $\text{T}_{\text{E}}\text{X}$ or $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ input was being translated.

An alternative to “programming” documents is to use a sophisticated editor such as that provided in Microsoft `Word` or Adobe `FrameMaker` to interactively enter and edit the document. (Editing operations allow the choice of fonts, selection of point size, inclusion of special symbols, and so on.) This approach to document preparation is called **WYSIWYG**—what you see is what you get—because the exact form of the document is always visible.

What are the relative advantages and disadvantages of the two approaches? Do analogues exist for ordinary programming languages?

6. Although compilers are designed to translate a particular language, they often allow calls to subprograms that are coded in some other language (typically, FORTRAN, C, or assembler). Why are such “foreign calls” allowed? In what ways do they complicate compilation?
7. Most C compilers (including the GNU `gcc` compiler) allow a user to examine the machine instructions generated for a given source program. Run the following program through such a C compiler and examine the instructions generated for the `for` loop. Next, recompile the program, enabling optimization, and reexamine the instructions generated for the `for` loop. What improvements have been made? Assuming that the program spends all of its time in the `for` loop, estimate the speedup obtained. Execute and time the two versions of the program and see how accurate your estimate is.

```
int proc(int a[]) {
    int sum = 0, i;
    for (i=0; i < 1000000; i++)
        sum += a[i];
    return sum;
}
```


8. C is sometimes called the “universal assembly language” in light of its ability to be very efficiently implemented on a wide variety of computer architectures. In light of this characterization, some compiler writers have chosen to generate C code, rather than a particular machine language, as their output. What are the advantages to this approach to compilation? Are there any disadvantages?
9. Many computer systems provide an interactive debugger (for example, `gdb` or `dbx`) to assist users in diagnosing and correcting run-time errors. Although a debugger is run long after a compiler has done its job, the two tools still must cooperate. What information (beyond the translation of a program) must a compiler supply to support effective run-time debugging?
10. Assume you have a source program P . It is possible to transform P into an equivalent program P' by reformatting P (by adding or deleting spaces, tabs, and line breaks), systematically renaming its variables (for example, changing all occurrences of `sum` to `total`), and reordering the definition of variables and subroutines.

Although P and P' are equivalent, they may well look very different. How could a compiler be modified to compare two programs and determine if they are equivalent (or very similar)? In what circumstances would such a tool be useful?

Bibliography

- [Ado90] Adobe Systems Incorporated. *PostScript Language Reference Manual, 2nd Edition*. Addison-Wesley, Reading, Mass., 1990.
- [App85] Andrew W. Appel. Semantics-directed code generation. *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana*, pages 315–324, 1985.
- [Coe89] David R. Coelho. *The VHDL Handbook*. Kluwer Academic Publishers, Boston, Mass., 1989.
- [Gri81] David Gries. *The Science of Programming*. Springer-Verlag, New York, N.Y., 1981.
- [Han85] Per Brinch Hansen. *Brinch Hansen on Pascal Compilers*. Prentice-Hall, 1985.
- [HJ92] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. *Winter Usenix Conference Proceedings*, pages 125–136, 1992.
- [HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [KHH91] Gerry Kane, Joe Heinrich, and Joseph Heinrich. *Mips Risc Architecture, Second Edition*. Prentice Hall, 1991.
- [KLP88] S. Kaufer, R. Lopez, and S. Pratap. Saber-c an interpreter-based programming environment for the c language. *Summer Usenix Conference Proceedings*, pages 161–171, 1988.
- [Knu68] Donald E. Knuth. Semantics of context-free languages. *Math. Systems Theory*, 2(2):127–145, 1968.
- [Lam95] Leslie Lamport. *L^AT_EX: A Document Preparation System*. Addison-Wesley Publishing Company, 1995.
- [McC65] John McCarthy. *Lisp 1.5 Programmer's Manual*. The MIT Press, Cambridge, Massachusetts, and London, England, 1965.

- [MTH90] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.
- [NN92] H.R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley and Sons, New York, N.Y., 1992.
- [Pal96] Samir Palnitkar. *Verilog HDL: A Guide to Digital Design and Synthesis*. Sun Microsystems Press, 1996.
- [RT88] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual, Third Edition*. Springer-Verlag, New York, N.Y., 1988.
- [Sch86] David A. Schmidt. *Denotational Semantics - A Methodology for Language Development*. Allyn and Bacon, 1986.
- [Sch97] Herbert Schildt. *Borland C++ : The Complete Reference*. Osborne McGraw-Hill, 1997.
- [Set83] Ravi Sethi. Control flow aspects of semantics-directed compiling. *ACM Transactions on Programming Languages and Systems*, 5(4), 1983.
- [Sta89] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, Inc., 1989.
- [Sun98] Sun Microsystems. Jdk 1.1.6 release notes., 1998.
- [Wan82] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3), 1982.
- [WG94] David L. Weaver and Tom Germond. *The Sparc Architecture Manual, Version 9*. Prentice Hall, 1994.
- [WG97] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 31–43, June 1997.
- [Wol96] Stephen Wolfram. *The Mathematica Book, Third Edition*. Cambridge University Press, 1996.

2

A Simple Compiler

In this chapter, we provide an overview of how the compilation process can be organized by considering in detail how a compiler can be built for a very small programming language. We begin in Section 2.1 by informally defining this language, which we call *ac*. In the rest of the chapter, we present the phases of a simple compiler for *ac*.

2.1 An Informal Definition of the *ac* Language

Our language is called *ac* (for *adding calculator*). It is a very simple language, yet it possesses components that are found in most programming languages. We use this language to examine the phases and data structures of a compiler. Following is the informal definition of *ac*.

- There are two data types: integer and float. An integer type is a decimal integer, as found in most programming languages. A float type allows five fractional digits after the decimal point.
- There are three reserved keywords: `f` (declares a float variable), `i` (declares an integer variable), and `p` (prints the value of a variable).
- The data type of an identifier is explicitly declared in the program. There are only 23 possible identifiers, drawn from the lowercase Roman alphabet. The exceptions are the three **reserved keywords** `f`, `i`, and `p`.
- When expressions are computed, conversion from integer type to float type is accomplished automatically. Conversion in the other direction is not allowed.

1	Prog	→	Dcls Stmts \$
2	Dcls	→	Dcl Dcls
3			λ
4	Dcl	→	floatdcl id
5			intdcl id
6	Stmts	→	Stmt Stmts
7			λ
8	Stmt	→	id assign Val ExprTail
9			print id
10	ExprTail	→	plus Val ExprTail
11			minus Val ExprTail
12			λ
13	Val	→	id
14			num

Figure 2.1: Context-free grammar for ac.

For the target of translation, we chose the Unix program `dc` (for *desk calculator*), which is a stacking (Reverse Polish) calculator. The target instructions must be acceptable to the `dc` program and faithfully represent the operations specified in an `ac` program. Compilation of `ac` to `dc` can be viewed as a study of larger systems, such as the portable Pascal and Java compilers, which produce a stack language as an intermediate representation [?].

2.2 Structure of an ac Compiler

The rest of this chapter presents a simple compiler for `ac`. The compiler's structure is based on the illustration in Figure Figure:one:onepointthree. Following are the main compilation steps.

1. The scanner reads a source program as a text file and produces a stream of tokens.
2. The parser processes tokens, determines the syntactic validity of the compiler's input, and creates an **abstract syntax tree** (AST) suitable for the compiler's subsequent activities.
3. The AST is walked to create a symbol table. This table associates type and other contextual information with the input program's variables.
4. Semantic checking is performed on the AST to determine the semantic validity of the compiler's input. The AST may be decorated with information required for subsequent compiler activities.
5. A translator walks the AST to generate `dc` code.

2.3 Formal Syntax Definition of ac

Before proceeding with creating the compiler's parts, in this section we specify the ac language more formally. For this, we use a **context-free grammar** (CFG) to specify the syntax of ac. The full grammar is shown in Figure 2.1. CFGs, first mentioned in Chapter Chapter:global:one, are discussed thoroughly in Chapter Chapter:global:four. Here, we view a CFG as a set of *rewriting rules*. A rewriting rule is also called a **production**. Following are two productions taken from the grammar for ac.

$$\begin{array}{l} \text{Stmt} \rightarrow \text{id assign Val ExprTail} \\ \quad \quad | \text{ print id} \end{array}$$

On its **left-hand side** (LHS), each production has exactly one symbol; on its **right-hand side** (RHS), it may have zero or more symbols. The symbols after the arrow or bar are a production's RHS. Stmt is the LHS of each production shown here. In a production, *any* occurrence of its LHS symbol can be replaced by the symbols on its RHS. The productions shown here specify that a Stmt can be replaced by two different strings of symbols. The top production lets a Stmt be rewritten as an assignment to an identifier, while the bottom production lets a Stmt be rewritten to specify the printing of an identifier's value.

Productions contains two kinds of symbols: *terminals* and *nonterminals*. A **terminal** is a grammar symbol that cannot be rewritten. Thus id and assign are symbols for which there are no productions specifying how they can be changed. The **nonterminal** symbols Val and ExprTail have productions that define how they can be rewritten. To ease readability in the grammar, we adopt the convention that nonterminals begin with an uppercase letter and terminals are all lowercase letters.

The purpose of a CFG is to specify a (typically infinite) set of legal token strings. A CFG does this in a remarkably elegant way. It starts with a single non-terminal symbol called the **start symbol**. Then it applies productions, rewriting nonterminals until only terminals remain. Any string of terminals that can be produced in this manner is considered syntactically valid. A string of terminals that *cannot* be produced by any sequence of nonterminal replacements is deemed illegal.

There are two special symbols that appear in the grammars of this text. Both symbols are regarded as terminal symbols, but they lie outside the normal terminal alphabet and cannot be supplied as input.

- The special symbol λ represents the **empty**, or **null**, string. When present, it appears as the only symbol on a production's RHS. In Rule 7 of Figure 2.1, λ indicates that the symbol Stmts can be replaced by *nothing*, effectively causing its erasure.
- The special symbol \$ represents termination of input. An input stream conceptually has an unbounded number of \$ symbols following the actual input.

Sentential Form	Production Number
<i>Prog</i>	
<u>Dcls</u> Stmt\$	1
<u>Dcl Dcls</u> Stmt\$	2
floatdcl id <u>Dcls</u> Stmt\$	4
floatdcl id <u>Dcl Dcls</u> Stmt\$	2
floatdcl id <u>intdcl id</u> <u>Dcls</u> Stmt\$	5
floatdcl id intdcl id <u>Stmts</u> \$	3
floatdcl id intdcl id <u>Stmt Stmt\$</u>	6
floatdcl id intdcl id id assign <u>Val ExprTail</u> Stmt\$	8
floatdcl id intdcl id id assign <u>num ExprTail</u> Stmt\$	14
floatdcl id intdcl id id assign num <u>Stmts</u> \$	12
floatdcl id intdcl id id assign num <u>Stmt Stmt\$</u>	6
floatdcl id intdcl id id assign num id assign <u>Val ExprTail</u> Stmt\$	8
floatdcl id intdcl id id assign num id assign id <u>ExprTail</u> Stmt\$	13
floatdcl id intdcl id id assign num id assign id <u>plus Val ExprTail</u> Stmt\$	10
floatdcl id intdcl id id assign num id assign id <u>plus num ExprTail</u> Stmt\$	14
floatdcl id intdcl id id assign num id assign id plus num <u>Stmts</u> \$	12
floatdcl id intdcl id id assign num id assign id plus num <u>Stmt Stmt\$</u>	6
floatdcl id intdcl id id assign num id assign id plus num <u>print id</u> Stmt\$	9
floatdcl id intdcl id id assign num id assign id plus num print id \$	7

Figure 2.2: Derivation of an ac program using the grammar in Figure 2.1.

Terminal	Regular Expression
floatdcl	f
intdcl	i
print	p
id	{a,b,c,...,z} - {f,i,p}
assign	=
plus	+
minus	-
num	{0,1,...,9} ⁺ {0,1,...,9} ⁺ .{0,1,...,9} ⁺

Figure 2.3: Formal definition of ac terminal symbols.

Grammars often generate a list of symbols from a nonterminal. Consider the productions for *Stmts*. They allow an arbitrary number of *Stmt* symbols to be produced. The first production for *Stmts* is recursive, and each use of this production generates another *Stmt*. The recursion is terminated by applying the second production for *Stmts*, thereby causing the final *Stmts* symbol to be erased.

To show how the grammar defines legal *ac* programs, the derivation of one such program is given in Figure 2.2, beginning with the start symbol *Prog*. Each line represents one step in the derivation. In each line, the leftmost nonterminal (in italics) is replaced by the underscored text shown on the next line. The right column shows the production number by which the derivation step is accomplished.

The derivation shown in Figure 2.2 explains how the terminal string is generated for the sample *ac* program. Although the CFG defines legal terminal sequences, how these terminals are “spelled” is another aspect of the language's definition. The terminal *assign* represents the assignment operator (typically spelled as `=` or `:=`). The terminal *id* represents an identifier. In most programming languages, an **identifier** is a word that begins with a letter and contains only letters and digits. In some languages, words such as `if`, `then`, and `while` are **reserved** and cannot be used as identifiers. It is the job of the *scanner* to recognize the occurrence of a string (such as `xy4z`) and return the corresponding token (*id*). Of course, the language designer must specify this correspondence.

Chapter Chapter:global:three defines *regular expressions* and shows how they can specify terminal spellings and automate the generation of scanners. By way of example, the regular expressions for terminals in *ac* are shown in Figure 2.3. The keywords are unusually terse so that the scanner can be readily constructed in Section 2.4. The specification given *id* allows any lowercase character except `f`, `i`, and `p`. The regular expression for *num* is the most complicated: The `|` symbol denotes choice, and the `+` symbol denotes repetition. Thus a *num* can be a sequence of digits *or* a sequence of digits followed by a decimal point followed by another sequence of digits. For the remainder of this chapter, we consider translation of the *ac* program shown in Figure 2.4. The figure also shows the sequence of terminal symbols that corresponds to the input program. The derivation shown textually in Figure 2.2 can be represented as a derivation (or parse) tree, also shown in Figure 2.4.

In the ensuing sections, we examine each step of the compilation process for the *ac* language, assuming an input that would produce the derivation shown in Figure 2.2. While the treatment is somewhat simplified, the goal is to show the activity and data structures of each phase.

2.4 The Scanner's Job

The scanner's job is to translate a stream of characters into a stream of *tokens*. A **token** represents an instance of a terminal symbol. Rigorous methods for constructing scanners based on regular expressions (such as those shown in Figure 2.3) are covered in Chapter Chapter:global:three. Here, we are content with

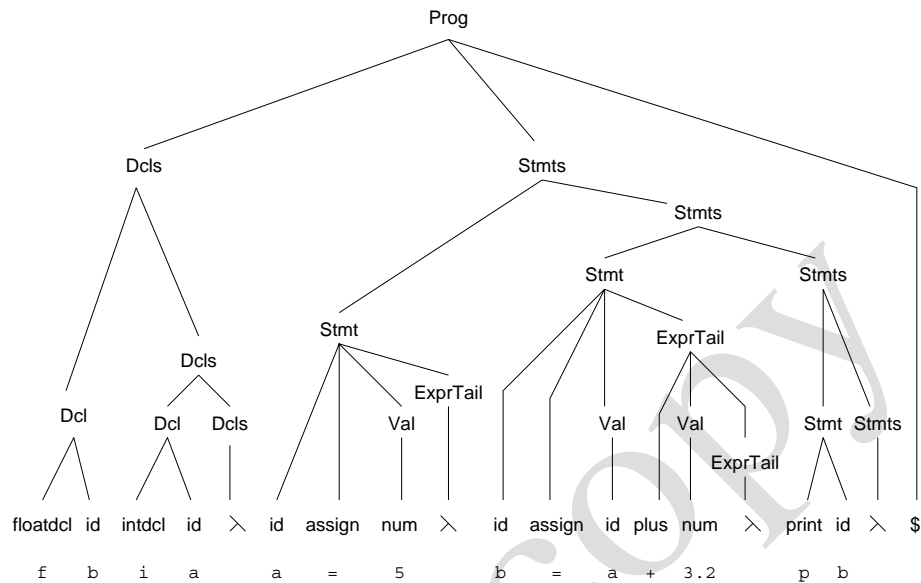


Figure 2.4: An ac program and its parse tree.

crafting an *ad hoc* scanner. While the automatic methods are more robust, the job at hand is sufficiently simple to attack manually.

Figure 2.5 shows the pseudocode for the basic scanner. This scanner examines an input stream and returns the stream's next token. As seen in this code, a token actually has two components, as follows.

- A token's *type* explains the token's membership in the terminal alphabet. All instances of a given terminal have the same token type.
- A token's *semantic value* provides additional information about the token.

For some tokens, such as plus and assign in ac, instance-specific information is unnecessary. Other tokens, such as id and num, require semantic information so that the compiler can record *which* identifier or number has been scanned. The code for scanning a number is the most complex, so it is relegated to the separate procedure SCANDIGITS. Although this scanner is written *ad hoc*, a principled approach supports its construction. The logic is patterned after the num token's regular expression. A recurring theme of this text is that the algorithms that enable automatic construction of a compiler's parts can often guide the manual construction of those parts.

The scanner, when called, must find the beginning of some token. Scanners are often instructed to ignore blanks and comments. Patterns for such input sequences are also specified using regular expressions, but the action triggered by such patterns does nothing. If the scanner encounters a character that cannot

```

function SCANNER(s) : Token
  if s.EOF( )
  then ans.type ← $
  else
    while s.PEEK( ) = blank do call s.ADVANCE( )
    if s.PEEK( ) ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    then
      ans.type ← num
      ans.val ← STRINGTOINT(SCANDIGITS( ))
    else
      ch ← s.ADVANCE( )
      switch (ch)
        case { a . . . z } – { i, f, p }
          ans.type ← id
          ans.val ← ch
        case f
          ans.type ← floatdcl
        case i
          ans.type ← intdcl
        case p
          ans.type ← print
        case =
          ans.type ← assign
        case +
          ans.type ← plus
        case -
          ans.type ← minus
        case default
          call LEXICALERROR( )
      return (ans)
    end
  end

```

Figure 2.5: Scanner for the ac language.

```

function SCANDIGITS(s) : String
    str ← “ ”
    while s.PEEK( ) ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} do
        str ← str + s.ADVANCE( )
    if s.PEEK( ) = “.”
    then
        str ← str + s.ADVANCE( )
        if s.PEEK( ) ∉ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
        then call ERROR( “Expected a digit” )
        while s.PEEK( ) ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} do
            str ← str + s.ADVANCE( )
    return (str)
end

```

Figure 2.6: Digit scanning for the ac language.

begin a token, then a **lexical error message** is issued; some scanners attempt to recover from such errors. A simple approach is to skip the offending character and continue scanning. This process continues until the beginning of *some* token is found. Once the scanner has found the beginning of a token, it then matches the longest possible character sequence that comprises a legal token.

Tokens come in many forms. Some are one character in length and cannot begin any other token. These are very easy to scan—a single character is read and the corresponding token is returned. Other tokens, such as + (in Java and C) are more difficult. When a + is seen, the next character must be inspected (but not yet read) to see if it extends the current token (for example, ++). Scanners can generally require a peek at the next character to determine whether the current token has been fully formed. For example, the end of a num token is signaled by the presence of a nondigit. Modular construction of the scanner is facilitated by allowing the next input character to be examined—perhaps many times—prior to consumption. This is typically accomplished by *buffering* the input. The method PEEK returns the contents of the buffer. The method ADVANCE returns the buffer's contents *and* advances the next character into the buffer.

Variable-length tokens, such as identifiers, literals, and comments, must be matched character-by-character. If the next character is part of the current token, it is consumed. When a character that cannot be part of the current token is reached, scanning is complete. When scanning is resumed, the last character inspected will be part of the next token.

2.5 The Parser's Job

The parser is responsible for determining if the stream of tokens provided by the scanner conforms to the language's grammar specification. In most compilers, the grammar serves not only to define the syntax of a programming language

```

procedure STMT(ts)
  if ts.PEEK( ) = id           1
  then
    call MATCH(ts, id)         2
    call MATCH(ts, assign)
    call VAL( )
    call EXPRTAIL( )
  else
    if ts.PEEK( ) = print     3
    then
      call MATCH(ts, print)
      call MATCH(ts, id)
    else call ERROR( )
end

```

Figure 2.7: Recursive-descent parsing procedure for Stmt.

but also to guide the automatic construction of a parser, as described in Chapters Chapter:global:five and Chapter:global:six. In this section, we build a parser for ac using a well-known parsing technique called *recursive descent*, which is described more fully in Chapter Chapter:global:five. We also consider a representation for the parsed program that serves as a record of the parse and as a means of conveying information between a compiler's components.

2.5.1 Recursive-Descent Parsing

Recursive descent is one of the simplest parsing techniques used in practical compilers. The name is taken from the recursive parsing routines that, in effect, descend through the derivation tree that is recognized as parsing proceeds. In recursive-descent parsing, each nonterminal *A* has an associated *parsing procedure*. This procedure must determine if a portion of the program's input contains a sequence of tokens derivable from *A*. The productions for *A* guide the parsing procedure as follows.

- The procedure contains conditionals that examine the next input token to predict which of *A*'s productions should be applied. If no production can be applied then an error message is issued. For the grammar shown in Figure 2.1, the parsing procedures associated with the nonterminals Stmt and Stmts are shown in Figures 2.7 and 2.8.
 - id predicts the production $\text{Stmt} \rightarrow \text{id assign Val ExprTail}$ and
 - print predicts the production $\text{Stmt} \rightarrow \text{print id}$.

The production is selected by the conditionals at Steps 1 and 3.

```

procedure STMTS(ts)
  if ts.PEEK( ) = id or ts.PEEK( ) = print 4
  then
    call STMT( )
    call STMTS( )
  else
    if ts.PEEK( ) = $
    then
      /* do nothing for λ-production */ 5
    else call ERROR( )
end

```

Figure 2.8: Recursive-descent parsing procedure for StmtS.

- Having selected a production $A \rightarrow \alpha$, the parsing procedure next recognizes the vocabulary symbols in α .
 - For terminal t , the procedure $\text{MATCH}(ts, t)$ is called to consume the next token in stream ts . If the token is not t then an error message is issued.
 - For a nonterminal B , the parsing procedure associated with B is invoked.

In Figure 2.7, the block of code at Step 2 determines that `id`, `assign`, `Val`, and `ExprTail` occur in sequence.

The parsing procedures can be recursive, hence the name “recursive descent.” In the full parser, each nonterminal has a procedure that is analogous to the ones shown in Figure Figure:two:RecDesStmt,RecDesStmts.

In our example, each production for `Stmt` begins with a distinct terminal symbol, so it is clear by inspection that `id` predicts `Stmt`'s first production and `print` predicts the second production. In general, an entire set of symbols can predict a given production; determining these **predict sets** can be more difficult.

- The production $\text{Stmts} \rightarrow \text{Stmt Stmts}$ begins with the nonterminal `Stmt`. This production is thus predicted by terminals that predict *any* production for `Stmt`. As a result, the predicate at Step 4 in Figure 2.8 checks for `id` or `print` as the next token.
- The production $\text{Stmts} \rightarrow \lambda$ offers no clues in its RHS as to which terminals predict this production. Grammar analysis can show that `$` predicts this production, because `$` is the only terminal that can appear *after* `Stmts`. No action is performed by the parser at Step 5 when applying the production $\text{Stmts} \rightarrow \lambda$, since there are no RHS symbols to match.

The grammar analysis needed to compute predict sets for an arbitrary CFG is discussed in Chapter Chapter:global:four.

Mechanically generated recursive-descent parsers can contain redundant tests for terminals. For example, the code in Figure 2.7 tests for the presence of an `id` at Steps 1 and 2. Modern software practice tolerates such redundancy if it simplifies or facilitates compiler construction. The redundancy introduced by one compiler component can often be eliminated by another, as discussed in Exercise 5.

Syntax errors arise when no production for the current nonterminal can be applied, or when a specific terminal fails to be matched. When such errors are discovered, a parser can be programmed to recover from the error and continue parsing. Ambitious parsers may even attempt to repair the error.

In practice, parsing procedures are rarely created *ad hoc*; they are based on the theory of *LL(k) parsing*, which is described in Chapter Chapter:global:five. Given a grammar for a language, the *LL(k)* method determines automatically if a suitable set of procedures can be written to parse strings in the grammar's language. Within each procedure, the conditionals that determine which production to apply must operate independently and without backtracking. Most tools that perform such analysis go one step further and automatically create the parsing code or its table representation.

2.5.2 Abstract Syntax Trees

What output should our recursive descent parser generate? It could generate a derivation tree corresponding to the tokens it has parsed. As Figure 2.4 shows, such trees can be rather large and detailed, even for very simple inputs. Moreover, CFGs often contain productions that serve only to *disambiguate* the grammar rather than to lend practical structure to the parsed inputs. In this section, we consider how to represent parsed inputs faithfully while avoiding unnecessary detail.

An **abstract syntax tree** (AST) contains the essential information in a derivation tree; unnecessary details are “abstracted out.” For example, an AST can elide inessential punctuation and delimiters (braces, semicolons, parentheses, and so on). An AST's design is also influenced by the needs of post-parsing activities. As a result, the AST's design is often revisited and modified during compiler construction. Chapter Chapter:global:seven considers the construction of ASTs in more detail. For our purposes, we place code in the recursive-descent parser to create and link the AST nodes into a tree. For example, our `Stmt` procedure becomes responsible for creating (and returning) a subtree that models the parsed `Stmt`.

The AST is designed to retain essential syntactic structure in a form that is amenable to subsequent processing. An AST for `ac` should be designed as follows.

- Declarations need not be retained in source form. However, a record of identifiers and their declared types must be retained to facilitate *symbol table construction* and *semantic type checking*, as described in Section 2.6.

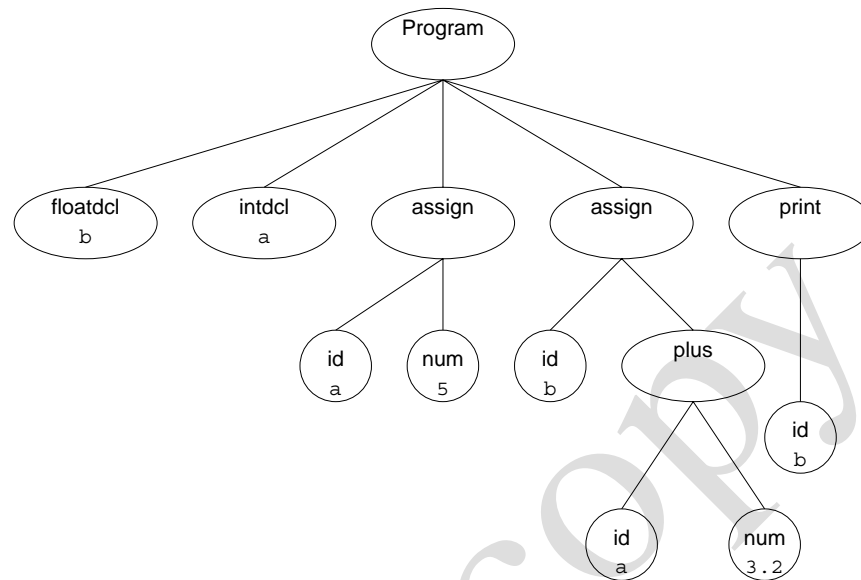


Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

- The order of the executable statements is important and must be explicitly represented, so that *code generation* (Section 2.7) can issue instructions in the proper order.
- An assignment statement must retain the identifier that will hold the computed value and the expression that computes the value.
- A print statement must retain the name of the identifier to be printed.

Figure 2.9 shows the AST resulting from the sample ac program. Cleaner and simpler than the derivation tree, the AST still contains all essential program structure and detail.

2.6 Semantic Analysis

After the parser builds a program's AST, the next step is **semantic analysis**, which is really a catch-all term for checks to ensure that the program truly conforms to a language's definition. Any aspect of the language's definition that cannot be readily formulated by grammar productions is fodder for semantic analysis, as follows.

- Declarations and scopes are processed to construct a *symbol table*.
- Language- and user-defined types are examined for consistency.

```

procedure SYMVISITNODE(n)
  if n.kind = floatdcl
  then call ENTERSYMBOL(n.id, float)
  else
    if n.kind = intdcl
    then call ENTERSYMBOL(n.id, integer)
  end
procedure ENTERSYMBOL(name, type)
  if SymbolTable[name] = undefined
  then SymbolTable[name] ← type
  else call ERROR(“duplicate declaration”)
end

```

Figure 2.10: Symbol table construction for ac.

- Operations and storage references are processed so that type-dependent behavior can become explicit in the program representation.

For example, in the assignment statement $x=y$, x is typically interpreted as an address and y is interpreted as a value. After semantic analysis, an AST identifier node always references the *address* of the identifier. Explicit operations are inserted into the AST to generate the value of an identifier from its address. As another example, some languages allow a single operator to have multiple meanings depending on the types of its operands. Such **operator overloading** greatly extends the notational power of a language. Semantic analysis uses type information to map an overloaded operator to its specific definition in context.

Most programming languages offer a set of **primitive types** that are available to all programs. Some programming languages allow additional types to be created, effectively extending the language's type system. In languages such as Java, C++, and Ada, semantic analysis is a significant task, considering the rules that must be enforced and the richness of the languages' primitive and extended types. The ensuing sections describe the comparatively simple semantic analysis for ac.

2.6.1 The Symbol Table

In ac, identifiers must be declared prior to use, but this requirement cannot be enforced at parse time. Thus, the first semantic-processing activity traverses the AST to record all identifiers and their types in a **symbol table**. Although the set of potential identifiers is infinite in most programming languages, we have simplified ac so that programs can mention at most 23 distinct identifiers. As a result, an ac symbol table has 23 entries, indicating each identifier's type: integer, float, or undefined. In most programming languages, the type information associated with a symbol includes other attributes, such as the identifier's scope, storage class, and protection.

Symbol	Type	Symbol	Type	Symbol	Type
a	integer	k		t	
b	float	l		u	
c		m		v	
d		n		w	
e		o		x	
g		q		y	
h		r		z	
j		s			

Figure 2.11: Symbol table for the ac program from Figure 2.4.

To create an ac symbol table, we traverse the AST, counting on the presence of a *declaration node* to trigger effects on the symbol table. In Figure 2.10, SYMVISITNODE shows the code to be applied as each node of the AST is visited. As declarations are discovered, ENTERSYMBOL checks that the given identifier has not been previously declared. Figure 2.11 shows the symbol table constructed for our example ac program. Blank entries in the table indicate that the given identifier is undefined.

2.6.2 Type Checking

Once symbol type information has been gathered, ac's executable statements can be examined for consistency of type usage. This process is called **type checking**. In an AST, expressions are evaluated starting at the leaves and proceeding upward to the root. To process nodes in this order, we perform type checking bottom-up over the AST. At each node, we apply SEMVISITNODE, shown in Figure 2.12, to ensure that operand types are either consistent or that a legal type conversion can be inserted to render the types consistent, as follows.

- For nodes that represent addition or subtraction, the computation is performed in float if either subtree has type float.
- For nodes representing the retrieval of a symbol's value, the type information is obtained by consulting the symbol table.
- Assignment demands that the type of the value match the type of the assignment's target.

Most programming language specifications include a **type hierarchy** that compares the language's types in terms of their generality. Our ac language follows in the tradition of Java, C, and C++, in which a float type is considered **wider** (*i.e.*, more general) than an integer. This is because every integer can be represented as a float. On the other hand, **narrowing** a float to an integer loses precision for some float values.

```

procedure SEMVISITNODE(n)
  switch (n.kind)
    case plus
      n.type ← CONSISTENT(n.child1, n.child2)
    case minus
      n.type ← CONSISTENT(n.child1, n.child2)
    case assign
      n.type ← CONVERT(n.child1, n.child2)
    case id
      n.type ← RETRIEVESYMBOL(n.name).type
    case num
      if CONTAINS DOT(n.name)
        then n.type ← float
        else n.type ← integer
  end
function CONSISTENT(c1, c2) : Type
  m ← GENERALIZE(c1.type, c2.type)
  if c1.type ≠ m
    then call CONVERT(c1, m)
  else
    if c2.type ≠ m
      then call CONVERT(c2, m)
  return (m)
end
function GENERALIZE(t1, t2) : Type
  if t1 = float or t2 = float
    then ans ← float
  else ans ← integer
  return (ans)
end
procedure CONVERT(n, t)
  if n.type = float and t = integer
    then call ERROR("Illegal type conversion")
  else
    if n.type = integer and t = float
      then /* replace node n by convert-to-float of node n */
      else /* nothing needed */
end

```

Figure 2.12: Type analysis for ac.

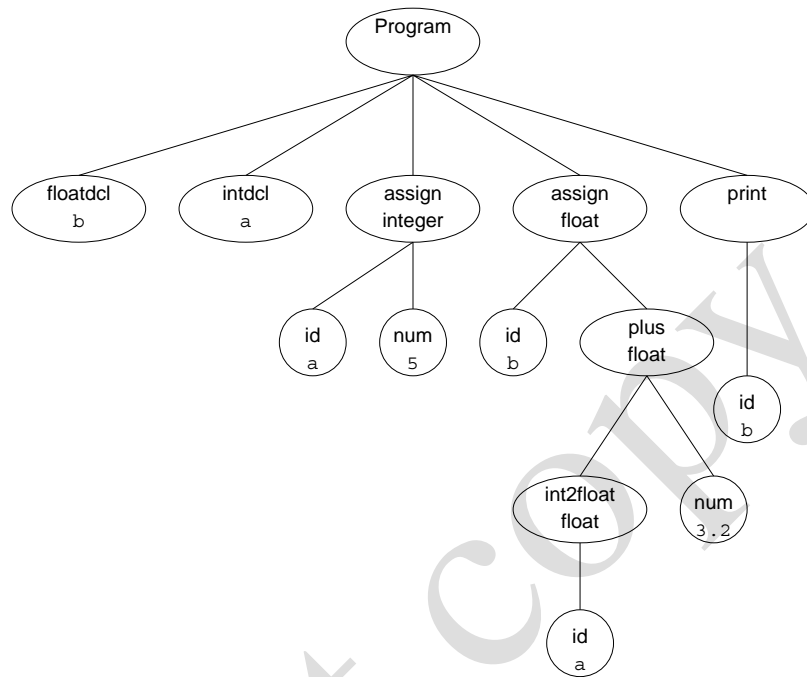


Figure 2.13: AST after semantic analysis.

Most languages allow automatic widening of type, so an integer can be converted to a float without the programmer having to specify this conversion explicitly. On the other hand, a float cannot become an integer in most languages unless the programmer explicitly calls for this conversion.

CONSISTENT, shown in Figure 2.12, is responsible for reconciling the type of a pair AST nodes using the following two steps.

1. The GENERALIZE function determines the least general (*i.e.*, simplest) type that encompasses its supplied pair of types. For ac, if either type is float, then float is the appropriate type; otherwise, integer will do.
2. The CONVERT procedure is charged with transforming the AST so that the ac program's *implicit* conversions become *explicit* in the AST. Subsequent compiler passes (particularly code generation) can assume a type-consistent AST in which all operations are explicit.

The results of applying semantic analysis to the AST of Figure 2.9 are shown in Figure 2.13.

2.7 Code Generation

With syntactic and semantic analysis complete, the final task undertaken by a compiler is the formulation of target-machine instructions that faithfully represent the semantics (*i.e.*, meaning) of the source program. This process is called **code generation**. Our translation exercise consists of generating code that is suitable for the dc program, which is a simple calculator based on a *stack machine* model. In a **stack machine**, most instructions receive their input from the contents at or near the top of an operand stack. The result of most instructions is pushed on the stack. Programming languages such as Pascal and Java are frequently translated into a portable, stack-machine representation [?].

Chapters Chapter:global:twelve, Chapter:global:thirteen, and Chapter:global:fifteen discuss code generation in detail. Automatic approaches generate code based on a description of the target machine. Such code generators greatly increase the portability of modern programming languages. Our translation task is sufficiently simple for an *ad hoc* approach. The AST has been suitably prepared for code generation by the insertion of type information. Such information is required for selecting the proper instructions. For example, most computers distinguish between instructions for float and integer data types.

We walk the AST starting at its root and let the nodes trigger code generation that is appropriate for their functions in the AST. The code generator shown in Figure 2.14 is recursive. In most programming languages, this conveniently accommodates the translation of AST constructs wherever they appear. The overall effect is to generate code for the program represented by the AST, as follows.

- For plus and minus, the code generator recursively generates code for the left and right subtrees. The resulting values are then at top-of-stack, and the appropriate operator is placed in the instruction stream to add or subtract the values.
- For assign, the value is computed and the result is stored in the appropriate dc register. The calculator's precision is then reset to integer by setting the fractional precision to zero; this is shown at Step 6 in Figure 2.14.
- Use of an id causes the value to be loaded from dc's register and pushed onto the stack.
- The print node is tricky because dc does not discard the value on top-of-stack after it is printed. The instruction sequence `s i` is generated at Step 7, thereby popping the stack and storing the value in dc's `i` register. Conveniently, the ac language precludes a program from using this register because the `i` token is reserved for spelling the terminal symbol integer.
- The change of type from integer to float at Step 8 requires setting dc's precision to five fractional decimal digits.

```

procedure CODEGEN(n)
  switch (n.kind)
    case Program
      foreach c ∈ Children(n) do call CODEGEN(c)
    case assign
      call CODEGEN(n.child2)
      call EMIT(“s”)
      call EMIT(n.child1.name)
      call EMIT(“0 k”)
    case plus
      call CODEGEN(n.child1)
      call CODEGEN(n.child2)
      call EMIT(“+”)
    case minus
      call CODEGEN(n.child1)
      call CODEGEN(n.child2)
      call EMIT(“-”)
    case id
      call EMIT(“1”)
      call EMIT(n.name)
    case print
      call CODEGEN(n.child1)
      call EMIT(“p”)
      call EMIT(“si”)
    case int2float
      call CODEGEN(n.child1)
      call EMIT(“5 k”)
    case num
      call EMIT(n.name)
  end

```

Figure 2.14: Code generation for ac

Code	Source	Comments
5 sa 0 k	a = 5	Push 5 on stack Store into the a register Reset precision to integer
1a 5 k 3.2 + sb 0 k	b = a + 3.2	Push the value of the a register Set precision to float Push 3.2 on stack Add Store result in the b register Reset precision to integer
1b p si	p b	Push the value of the b register Print the value Pop the stack by storing into the i register

Figure 2.15: Code generated for the AST shown in Figure 2.9.

Figure 2.15 shows how code is generated for the AST shown in Figure 2.9. The horizontal lines delineate the code generated in turn for each child of the AST's root. Even in this *ad hoc* code generator, one can see principles at work. The code sequences triggered by various AST nodes dovetail to carry out the instructions of the input program. Although the task of code generation for real programming languages and targets is more complex, the theme still holds that pieces of individual code generation contribute to a larger effect.

This finishes our tour of a compiler for the ac language. While each of the phases becomes more involved as we move toward working with real programming languages, the spirit of each phase remains the same. In the ensuing chapters, we discuss how to automate many of the tasks described in this chapter. We develop the skills necessary to craft a compiler's phases to accommodate issues that arise when working with real programming languages.

Exercises

1. The CFG shown in Figure 2.1 defines the syntax of ac programs. Explain how this grammar enables you to answer the following questions.
 - (a) Can an ac program contain only declarations (and no statements)?
 - (b) Can a print statement precede all assignment statements?
2. Sometimes it is necessary to modify the syntax of a programming language. This is done by changing the CFG that the language uses. What changes would have to be made to ac's CFG (Figure 2.1) to implement the following changes?
 - (a) All ac programs must contain at least one statement.
 - (b) All integer declarations must precede all float declarations.
 - (c) The first statement in any ac program must be an assignment statement.
3. Extend the ac scanner (Figure 2.5) so that the following occurs.
 - (a) A floatdcl can be represented as either `f` or `float`. (That is, a more Java-like declaration may be used.)
 - (b) An intdcl can be represented as either `i` or `int`.
 - (c) A num may be entered in exponential (scientific) form. That is, an ac num may be suffixed with an optionally signed exponent (`1.0e10`, `123e-22` or `0.31415926535e1`).
4. Write the recursive-descent parsing procedures for all nonterminals in Figure 2.1.
5. The recursive-descent code shown in Figure 2.7 contains redundant tests for the presence of some terminal symbols. Show how these tests can be eliminated.
6. In ac, as in many computer languages, variables are considered *uninitialized* after they are declared. A variable needs to be given a value (in an assignment statement) before it can be correctly used in an expression or print statement.

Suggest how to extend ac's semantic analysis (Section 2.6) to detect variables that are used before they are properly initialized.
7. Implement semantic actions in the recursive-descent parser for ac to construct ASTs using the design guidelines in Section 2.5.2.
8. The grammar for ac shown in Figure 2.1 requires all declarations to precede all executable statements. In this exercise, the ac language is extended so that declarations and executable statements can be interspersed. However, an identifier cannot be mentioned in an executable statement until it has been declared.

- (a) Modify the CFG in Figure 2.1 to accommodate this language extension.
 - (b) Discuss any revisions you would consider in the AST design for ac.
 - (c) Discuss how semantic analysis is affected by the changes you envision for the CFG and the AST.
9. The code in Figure 2.10 examines an AST node to determine its effect on the symbol table. Explain why the order in which nodes are visited does or does not matter with regard to symbol-table construction.

Do not copy

3

Scanning—Theory and Practice

In this chapter, we discuss the theoretical and practical issues involved in building a scanner. In Section 3.1, we give an overview of how a scanner operates. In Section 3.2, we introduce a declarative *regular expression* notation that is well-suited to the formal definition of tokens. In Section 3.3, the correspondence between regular expressions and *finite automata* is studied. Finite automata are especially useful because they are procedural in nature and can be directly executed to read characters and group them into tokens. As a case study, a well-known scanner generator, *Lex*, is considered in some detail in Section 3.4. *Lex* takes token definitions (in a declarative form—regular expressions) and produces a complete scanner subprogram, ready to be compiled and executed. Section 3.5 briefly considers other scanner generators.

In Section 3.6, we discuss the practical considerations needed to build a scanner and integrate it with the rest of the compiler. These considerations include anticipating the tokens and contexts that may complicate scanning, avoiding performance bottlenecks, and recovering from lexical errors. We conclude the chapter with Section 3.7, which explains how scanner generators, such as *Lex*, translate regular expressions into finite automata and how finite automata may be converted to equivalent regular expressions. Readers who prefer to view a scanner generator as simply a black box may skip this section. However, the material does serve to reinforce the concepts of regular expressions and finite automata introduced in earlier sections. The section also illustrates how finite automata can be built, merged, simplified, and even optimized.

3.1 Overview of a Scanner

The primary function of a scanner is to transform a character stream into a token stream. A scanner is sometimes called a **lexical analyzer**, or **lexer**. The names “scanner,” “lexical analyzer,” and “lexer” are used interchangeably. The scanner discussed in Chapter 2 was simple and could easily be coded by any competent programmer. In this chapter, we develop a thorough and systematic approach to scanning that will allow us to create scanners for complete programming languages.

We introduce formal notations for specifying the precise structure of tokens. At first glance, this may seem unnecessary because of the simple token structure found in most programming languages. However, token structure can be more detailed and subtle than one might expect. For example, consider simple quoted strings in C, C++, and Java. The body of a string can be any sequence of characters except a quote character, which must be escaped. But is this simple definition really correct? Can a newline character appear in a string? In C it cannot, unless it is escaped with a backslash. Doing this avoids a “runaway string” that, lacking a closing quote, matches characters intended to be part of other tokens. While C, C++, and Java allow escaped newlines in strings, Pascal forbids them. Ada goes further still and forbids all unprintable characters (precisely because they are normally unreadable). Similarly, are null (zero-length) strings allowed? C, C++, Java, and Ada allow them, but Pascal forbids them. In Pascal, a string is a packed array of characters and zero-length arrays are disallowed.

A precise definition of tokens is necessary to ensure that lexical rules are clearly stated and properly enforced. Formal definitions also allow a language designer to anticipate design flaws. For example, virtually all languages allow fixed decimal numbers, such as 0.1 and 10.01. But should .1 or 10. be allowed? In C, C++, and Java, they are. But in Pascal and Ada they are not—and for an interesting reason. Scanners normally seek to match as many characters as possible so that, for example, ABC is scanned as one identifier rather than three. But now consider the character sequence 1..10. In Pascal and Ada, this should be interpreted as a range specifier (1 to 10). However, if we were careless in our token definitions, we might well scan 1..10 as two real literals, 1. and .10, which would lead to an immediate (and unexpected) syntax error. (The fact that two real literals *cannot* be adjacent is reflected in the **context-free grammar** (CFG), which is enforced by the parser, not the scanner.)

When a formal specification of token and program structure is given, it is possible to examine a language for design flaws. For example, we could analyze all pairs of tokens that can be adjacent to each other and determine whether the two if catenated might be incorrectly scanned. If so, a separator may be required. In the case of adjacent identifiers and reserved words, a blank space (whitespace) suffices to distinguish the two tokens. Sometimes, though, the lexical or program syntax might need to be redesigned. The point is that language design is far more involved than one might expect, and formal specifications allow flaws to be discovered before the design is completed.

All scanners, independent of the tokens to be recognized, perform much the same function. Thus writing a scanner from scratch means reimplementing components that are common to all scanners; this means a significant duplication of effort. The goal of a **scanner generator** is to limit the effort of building a scanner to that of specifying which tokens the scanner is to recognize. Using a formal notation, we tell the scanner generator what tokens we want recognized. It then is the generator's responsibility to produce a scanner that meets our specification. Some generators do not produce an entire scanner. Rather, they produce tables that can be used with a standard driver program, and this combination of generated tables and standard driver yields the desired custom scanner.

Programming a scanner generator is an example of **declarative programming**. That is, unlike in ordinary, or **procedural programming**, we do not tell a scanner generator *how* to scan but simply *what* to scan. This is a higher-level approach and in many ways a more natural one. Much recent research in computer science is directed toward declarative programming styles; examples are database query languages and Prolog, a “logic” programming language. Declarative programming is most successful in limited domains, such as scanning, where the range of implementation decisions that must be made automatically is limited. Nonetheless, a long-standing (and as yet unrealized) goal of computer scientists is to automatically generate an entire production-quality compiler from a specification of the properties of the source language and target computer.

Although our primary focus in this book is on producing correct compilers, performance is sometimes a real concern, especially in widely used “production compilers.” Surprisingly, even though scanners perform a simple task, they can be significant performance bottlenecks if poorly implemented. This because scanners must wade through the text of a program character-by-character.

Suppose we want to implement a very fast compiler that can compile a program in a few seconds. We'll use 30,000 lines a minute (500 lines a second) as our goal. (Compilers such as “Turbo C++” achieve such speeds.) If an average line contains 20 characters, the compiler must scan 10,000 characters per second. On a 10 MIPS processor (10,000,000 instructions executed per second), even if we did nothing but scanning, we'd have only 1,000 instructions per input character to spend. But because scanning isn't the only thing a compiler does, 250 instructions per character is more realistic. This is a rather tight budget, considering that even a simple assignment takes several instructions on a typical processor. Although multi-MIPS processors are common these days and 30,000 lines per minute is an ambitious speed, clearly a poorly coded scanner can dramatically impact a compiler's performance.

3.2 Regular Expressions

Regular expressions are a convenient way to specify various simple (although possibly infinite) sets of strings. They are of practical interest because they can specify the structure of the tokens used in a programming language. In particular, you can use regular expressions to program a scanner generator.

Regular expressions are widely used in computer applications other than compilers. The Unix utility `grep` uses them to define search patterns in files. Unix shells allow a restricted form of regular expressions when specifying file lists for a command. Most editors provide a “context search” command that enables you to specify desired matches using regular expressions.

A set of strings defined by regular expressions is called a **regular set**. For purposes of scanning, a token class is a regular set whose structure is defined by a regular expression. A particular instance of a token class is sometimes called a **lexeme**; however, we simply call a string in a token class an *instance* of that token. For example, we call the string `abc` an identifier if it matches the regular expression that defines the set of valid identifier tokens.

Our definition of regular expressions starts with a finite character set, or **vocabulary** (denoted Σ). This vocabulary is normally the character set used by a computer. Today, the *ASCII* character set, which contains 128 characters, is very widely used. Java, however, uses the *Unicode* character set. This set includes all of the ASCII characters as well as a wide variety of other characters.

An empty, or null, string is allowed (denoted λ). This symbol represents an empty buffer in which no characters have yet been matched. It also represents an optional part of a token. Thus an integer literal may begin with a plus or minus, or, if it is unsigned, it may begin with λ .

Strings are built from characters in the character set Σ via *catenation* (that is, by joining individual characters to form a string). As characters are catenated to a string, it grows in length. For example, the string `do` is built by first catenating `d` to λ and then catenating `o` to the string `d`. The null string, when catenated with any string s , yields s . That is, $s\lambda \equiv \lambda s \equiv s$. Catenating λ to a string is like adding 0 to an integer—nothing changes.

Catenation is extended to sets of strings as follows. Let P and Q be sets of strings. The symbol \in represents set membership. If $s_1 \in P$ and $s_2 \in Q$, then string $s_1s_2 \in (PQ)$. Small finite sets are conveniently represented by listing their elements, which can be individual characters or strings of characters. Parentheses are used to delimit expressions, and $|$, the alternation operator, is used to separate alternatives. For example, D , the set of the ten single digits, is defined as $D = (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)$. (In this text, we often use abbreviations such as $(0 | \dots | 9)$ rather than enumerate a complete list of alternatives. The “...” symbol is *not* part of our regular expression notation.)

A **meta-character** is any punctuation character or regular expression operator. A meta-character must be quoted when used as an ordinary character in order to avoid ambiguity. (Any character or string may be quoted, but unnecessary quotation is avoided to enhance readability.) The characters `(`, `)`, `'`, `*`, `+`, and `|` are meta-characters. For example the expression `('(' | ')') | ; | ,` defines four single character tokens (left parenthesis, right parenthesis, semicolon, and comma) that we might use in a programming language. The parentheses are quoted to show they are meant to be individual tokens and not delimiters in a larger regular expression.

Alternation can be extended to sets of strings. Let P and Q be sets of strings. Then string $s \in (P | Q)$ if, and only if, $s \in P$ or $s \in Q$. For example, if LC is

the set of lowercase letters and UC is the set of uppercase letters, then $(LC | UC)$ denotes the set of all letters (in either case).

Large (or infinite) sets are conveniently represented by operations on finite sets of characters and strings. Catenation and alternation may be used. A third operation, **Kleene closure**, as defined below, is also allowed. The operator $*$ is the postfix *Kleene closure operator*. Here's an example. Let P be a set of strings. Then P^* represents all strings formed by the catenation of zero or more selections (possibly repeated) from P . (Zero selections are represented by λ .) For example, LC^* is the set of all words composed only of lowercase letters and of any length (including the zero-length word, λ).

Precisely stated, a string $s \in P^*$ if, and only if, s can be broken into zero or more pieces: $s = s_1s_2\dots s_n$ such that each $s_i \in P$ ($n \geq 0, 1 \leq i \leq n$). We explicitly allow $n = 0$ so that λ is always in P^* .

Now that we've introduced the operators used in regular expressions, we can define regular expressions as follows.

- \emptyset is a regular expression denoting the empty set (the set containing no strings). \emptyset is rarely used but is included for completeness.
- λ is a regular expression denoting the set that contains only the empty string. This set is not the same as the empty set because it does contain one element.
- A string s is a regular expression denoting a set containing the single string s . If s contains meta-characters, s can be quoted to avoid ambiguity.
- If A and B are regular expressions, then $A | B$, AB , and A^* are also regular expressions. They denote, respectively, the alternation, catenation, and Kleene closure of the corresponding regular sets.

Each regular expression denotes a regular set. Any finite set of strings can be represented by a regular expression of the form $(s_1 | s_2 | \dots | s_k)$. Thus the reserved words of ANSI C can be defined as $(\text{auto} | \text{break} | \text{case} | \dots)$.

The following additional operations are also useful. They are not strictly necessary because their effect can be obtained (perhaps somewhat clumsily) using the three standard regular operators (alternation, catenation, and Kleene closure).

- P^+ , sometimes called **positive closure**, denotes all strings consisting of one or more strings in P catenated together: $P^+ = (P^* | \lambda)$ and $P^+ = P P^*$. For example, the expression $(0 | 1)^+$ is the set of all strings containing one or more bits.
- If A is a set of characters, $\text{Not}(A)$ denotes $(\Sigma - A)$, that is, all *characters* in Σ not included in A . Since $\text{Not}(A)$ can never be larger than Σ and Σ is finite, $\text{Not}(A)$ must also be finite. It therefore is regular. $\text{Not}(A)$ does not contain λ because λ is not a character (it is a zero-length string). As an example, $\text{Not}(\text{Eol})$ is the set of all characters excluding Eol (the end-of-line character; in Java or C, $\backslash n$).

It is possible to extend `Not()` to strings, rather than just Σ . If S is a set of strings, we can define \bar{S} to be $(\Sigma^* - S)$, that is, the set of all strings except those in S . Although \bar{S} is usually infinite, it also is regular if S is. (See Exercise 18.)

- If k is a constant, the set A^k represents all strings formed by concatenating k (possibly different) strings from A . That is, $A^k = (AAA \dots)$ (k copies). Thus $(0 | 1)^{32}$ is the set of all bit strings exactly 32 bits long.

3.2.1 Examples

Next, we explore how regular expressions can be used to specify tokens. Let D be the set of the ten single digits and L be the set of all letters (52 in all). Then the following is true.

- A Java or C++, single-line comment that begins with `//` and ends with `Eol` can be defined as

$$\textit{Comment} = // \text{Not}(\text{Eol})^* \text{Eol}$$

This regular expression says that a comment begins with two slashes and ends at the *first* end-of-line. Within the comment, any sequence of characters is allowed that does not contain an end-of-line. (This guarantees that the first end-of-line we see ends the comment.)

- A fixed-decimal literal (for example, 12.345) can be defined as

$$\textit{Lit} = D^+ . D^+$$

One or more digits must be on both sides of the decimal point, so this definition excludes `.12` and `35.`

- An optionally signed integer literal can be defined as

$$\textit{IntLiteral} = ('+' | '-' | \lambda) D^+$$

An integer literal is one or more digits preceded by a plus or minus or no sign at all (λ). So that the plus sign is not confused with the Kleene closure operator, it is quoted.

- A more complicated example is a comment delimited by `##` markers, which allows single `#`'s within the comment body:

$$\textit{Comment2} = ## ((\# | \lambda) \text{Not}(\#))^* ##$$

Any `#` that appears within this comment's body must be followed by a non-`#` so that a premature end of comment marker, `##`, is not found.

All *finite* sets are regular. However, some but not all *infinite* sets are regular. For example, consider the set of balanced brackets of the form $[[[\dots]]]$. This set is defined formally as $\{[{}^m]{}^m \mid m \geq 1\}$. This is a set that is known not to be regular. The problem is that any regular expression that tries to define it either does not get all balanced nestings or includes extra, unwanted strings. (Exercise 14 proves this.)

It is easy to write a CFG that defines balanced brackets precisely. In fact, all regular sets can be defined by CFGs. Thus, the bracket example shows that CFGs are a more powerful descriptive mechanism than regular expressions. Regular expressions are, however, quite adequate for specifying token-level syntax. Moreover, for every regular expression we can create an efficient device, called a *finite automaton*, that recognizes exactly those strings that match the regular expression's pattern.

3.3 Finite Automata and Scanners

A **finite automaton** (FA) can be used to recognize the tokens specified by a regular expression. An FA (plural: finite automata) is a simple, idealized computer that recognizes strings as belonging to regular sets. An FA consists of the following:

- A finite set of *states*
- A finite *vocabulary*, denoted Σ
- A set of *transitions* (or *moves*) from one state to another, labeled with characters in Σ
- A special state called the *start* state
- A subset of the states called the *accepting*, or *final*, states

These components of an FA can be represented graphically as shown in Figure 3.1.

An FA also can be represented graphically using a **transition diagram**, composed of the components shown in Figure 3.1. Given a transition diagram, we begin at the start state. If the next input character matches the label on a transition from the current state, we go to the state to which it points. If no move is possible, we stop. If we finish in an accepting state, the sequence of characters read forms a valid token; otherwise, a valid token has not been seen. In the transition diagram shown in Figure 3.2, the valid tokens are the strings described by the regular expression $(ab(c)^*)^*$.

As an abbreviation, a transition may be labeled with more than one character (for example, $\text{Not}(c)$). The transition may be taken if the current input character matches any of the characters labeling the transition.

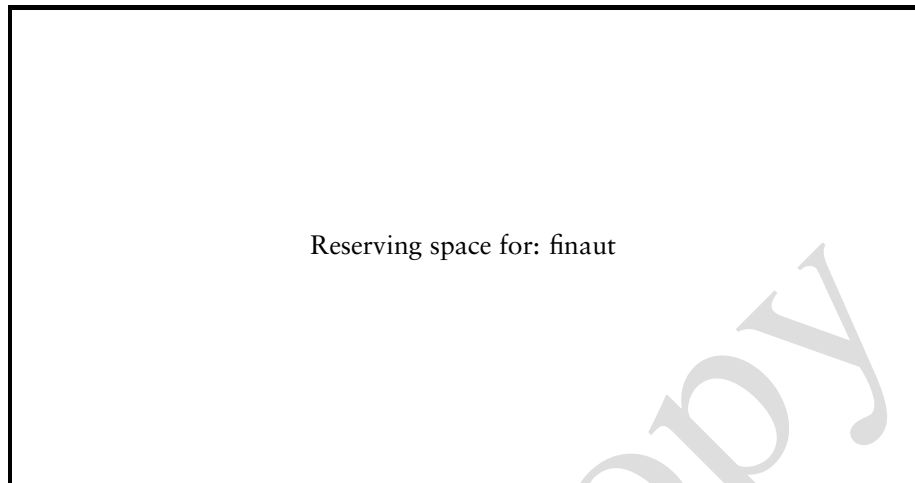


Figure 3.1: The four parts of a finite automation.

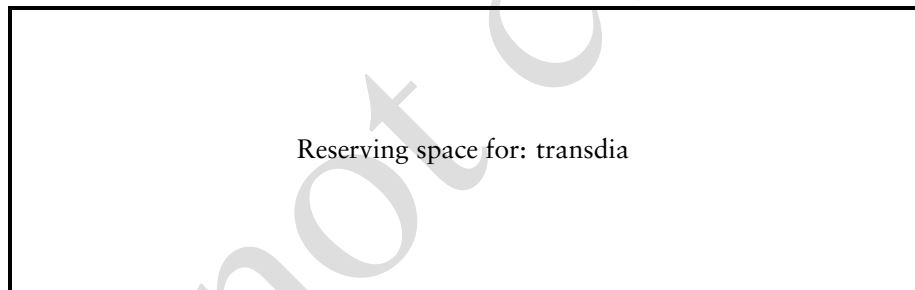


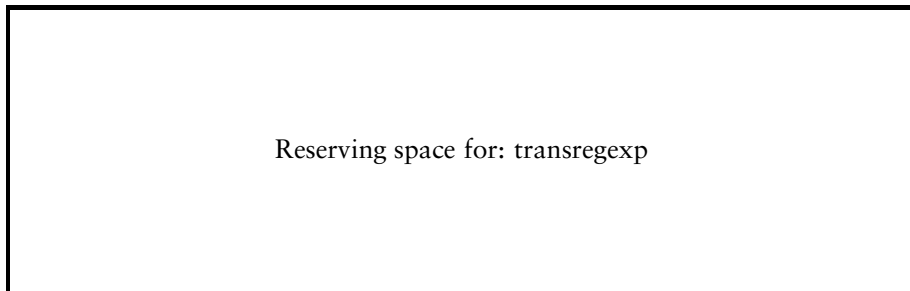
Figure 3.2: Transition diagram of an FA.

3.3.1 Deterministic Finite Automata

An FA that always has a *unique* transition (for a given state and character) is a **deterministic finite automation** (DFA). DFAs are easy to program and are often used to drive a scanner. A DFA is conveniently represented in a computer by a **transition table**. A transition table, T , is a two-dimensional array indexed by a DFA state and a vocabulary symbol. Table entries are either a DFA state or an error flag (often represented as a blank table entry). If we are in state s and read character c , then $T[s,c]$ will be the next state we visit, or $T[s,c]$ will contain an error flag indicating that c cannot extend the current token. For example, the regular expression

```
// Not(Eol)*Eol
```

which defines a Java or C++ single-line comment, might be translated as shown in Figure 3.3. The corresponding transition table is shown in Figure 3.4.

Figure 3.3: Translation of the regular expression `// Not(Eol)*Eol`.

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 3.4: Transition table of the regular expression `// Not(Eol)*Eol`.

A full transition table will contain one column for each character. To save space, *table compression* is sometimes utilized. In that case, only nonerror entries are explicitly represented in the table. This is done by using hashing or linked structures.

Any regular expression can be translated into a DFA that accepts (as valid tokens) the set of strings denoted by the regular expression. This translation can be done manually by a programmer or automatically by a scanner generator.

Coding the DFA

A DFA can be coded in one of two forms:

1. Table-driven
2. Explicit control

In the *table-driven* form, the transition table that defines a DFA's actions is explicitly represented in a run-time table that is “interpreted” by a driver program. In the *explicit control form*, the transition table that defines a DFA's actions appears implicitly as the control logic of the program. Typically, individual program statements correspond to distinct DFA states. For example, suppose *CurrentChar* is the current input character. End-of-file is represented by a special character value, *Eof*. Using the DFA for the Java comments illustrated previously, the two approaches would produce the programs illustrated in Figure 3.5 and 3.6.

```

/* Assume CurrentChar contains the first character to be scanned */
State ← StartState
while true do
    NextState ← T[State, CurrentChar]
    if NextState = error
    then break
    State ← NextState
    READ( CurrentChar)
if State ∈ AcceptingStates
then /* Return or process valid token */
else /* signal a lexical error */

```

Figure 3.5: Scanner driver interpreting a transition table.

```

/* Assume CurrentChar contains the first character to be scanned */
if CurrentChar = 'l'
then
    READ( CurrentChar)
    if CurrentChar = 'l'
    then
        repeat
            READ( CurrentChar)
        until CurrentChar ∈ { Eol, Eof }
    else /* Signal a lexical error */
else /* Signal a lexical error */
if CurrentChar = Eol
then /* Return or process valid token */
else /* Signal a lexical error */

```

Figure 3.6: Explicit control scanner.

The table-driven form is commonly produced by a scanner generator; it is token-independent. It uses a simple driver that can scan any token, provided the transition table is properly stored in T . The explicit control form may be produced automatically or by hand. The token being scanned is “hardwired” into the code. This form of a scanner is usually easy to read and often is more efficient, but it is specific to a single token definition.

Following are two more examples of regular expressions and their corresponding DFAs.

1. A FORTRAN-like real literal (which requires either digits on either or both sides of a decimal point or just a string of digits) can be defined as

$$RealLit = (D^+ (\lambda | \cdot | \cdot)) | (D^* \cdot D^+)$$

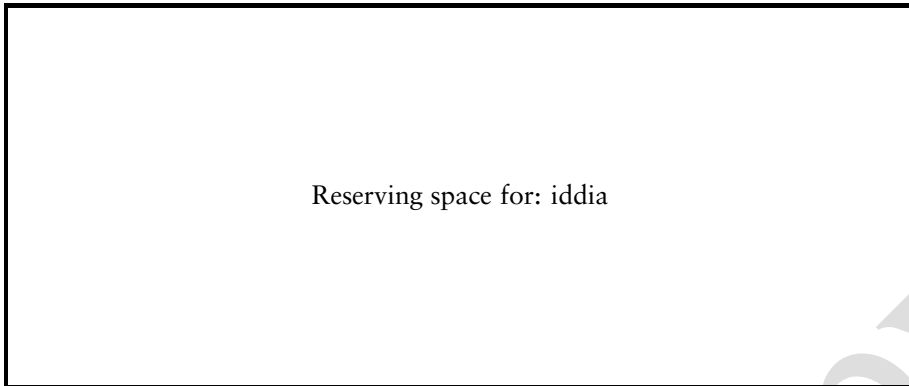


Figure 3.7: DFA for FORTRAN-like real literal.

which corresponds to the DFA shown in Figure 3.7.

2. An identifier consisting of letters, digits, and underscores. It begins with a letter and allows no adjacent or trailing underscores. It may be defined as

$$ID = L (L | D)^* (-(L | D)^*)^+$$

This definition includes identifiers such as `sum` or `unit_cost` but excludes `_one` and `two_` and `grand_total`. The corresponding DFA is shown in Figure 3.8.

Transducers

So far, we haven't saved or processed the characters we've scanned—they've been matched and then thrown away. It is useful to add an output facility to an FA; this makes the FA a **transducer**. As characters are read, they can be transformed and catenated to an output string. For our purposes, we limit the transformation operations to saving or deleting input characters. After a token is recognized, the transformed input can be passed to other compiler phases for further processing. We use the notation shown in Figure 3.9. For example, for Java and C++ comments, we might write the DFA shown in Figure 3.10. A more interesting example is given by Pascal-style quoted strings, according to the regular expression

$$(" (Not(" | "")^* ").$$

A corresponding transducer might be as shown in Figure 3.11. The input `""Hi""` would produce output `Hi`.

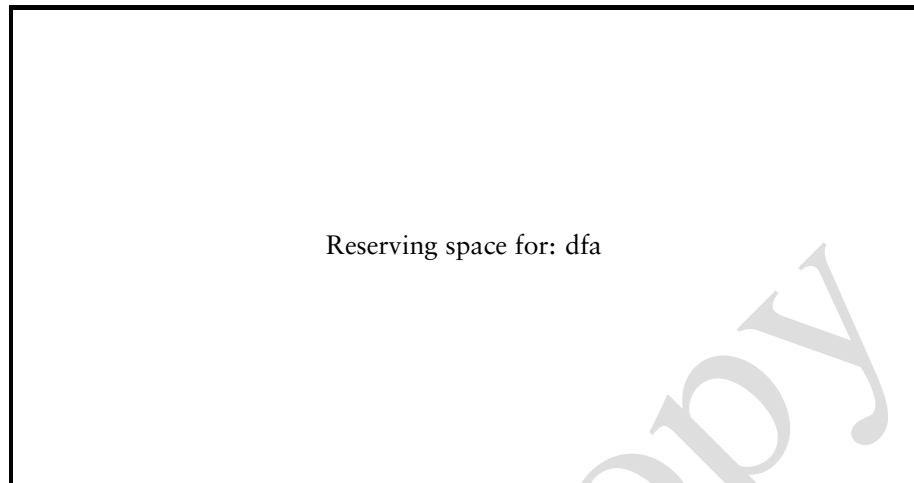


Figure 3.8: DFA for identifiers with underscores.

3.4 The Lex Scanner Generator

Next, as a case study in the design of scanner generation tools, we discuss a very popular scanner generator, Lex. Later, we briefly discuss several other scanner generators.

Lex was developed by M. E. Lesk and E. Schmidt of AT&T Bell Laboratories. It is used primarily with programs written in C or C++ running under the Unix operating system. Lex produces an entire scanner module, coded in C, that can be compiled and linked with other compiler modules. A complete description of Lex can be found in [LS75] and [LMB92]. Flex (see [Pax88]) is a widely used, freely distributed reimplement of Lex that produces faster and more reliable scanners. Valid Lex scanner specifications may, in general, be used with Flex without modification.

The operation of Lex is illustrated in Figure 3.12. Here are the steps:

1. A scanner specification that defines the tokens to be scanned and how they are to be processed is presented to Lex.
2. Lex then generates a complete scanner coded in C.
3. This scanner is compiled and linked with other compiler components to create a complete compiler.

Using Lex saves a great deal of effort programming a scanner. Many low-level details of the scanner (reading characters efficiently, buffering them, matching characters against token definitions, and so on) need not be explicitly programmed. Rather, we can focus on the character structure of tokens and how they are to be processed.

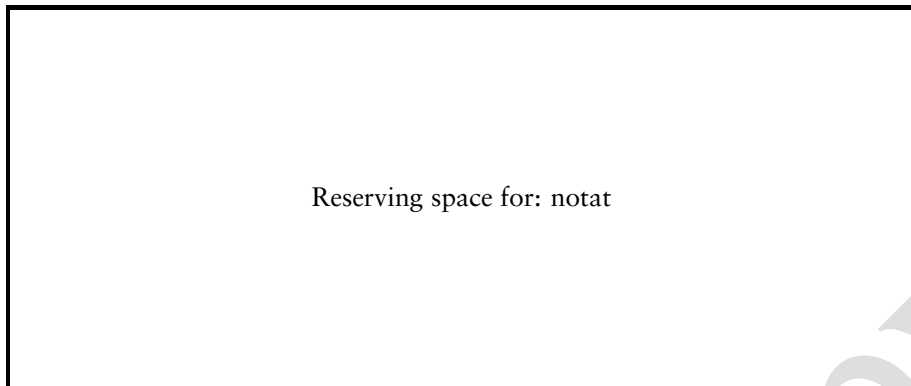


Figure 3.9: Transducer notation.

The primary purpose of this section is to show how regular expressions and related information are presented to scanner generators. A good way to learn how to use Lex is to start with the simple examples presented here and then gradually generalize them to solve the problem at hand. To inexperienced readers, Lex's rules may seem unnecessarily complex. It is best to keep in mind that the key is always the specification of tokens as regular expressions. The rest is there simply to increase efficiency and handle various details.

3.4.1 Defining Tokens in Lex

Lex's approach to scanning is simple. It allows the user to associate regular expressions with commands coded in C (or C++). When input characters that match the regular expression are read, the command is executed. As a user of Lex, you don't need to tell it *how* to match tokens. You need only tell it *what* you want done when a particular token is matched.

Lex creates a file `lex.yy.c` that contains an integer function `yyllex()`. This function is normally called from the parser whenever another token is needed. The value that `yyllex()` returns is the token code of the token scanned by Lex. Tokens such as whitespace are deleted simply by having their associated command not return anything. Scanning continues until a command with a return in it is executed.

Figure 3.13 illustrates a simple Lex definition for the three reserved words—`f`, `i`, and `p`—of the `ac` language introduced in Chapter `Chapter:global:two`. When a string matching any of these three reserved keywords is found, the appropriate token code is returned. It is vital that the token codes that are returned when a token is matched are identical to those expected by the parser. If they are not, the parser won't “see” the same token sequence produced by the scanner. This will cause the parser to generate false syntax errors based on the incorrect token stream it sees.

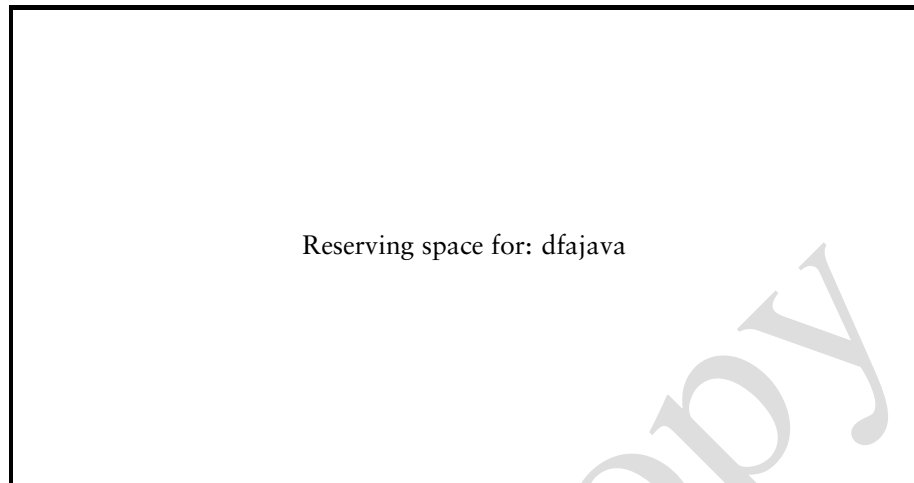


Figure 3.10: The DFA for Java and C++ comments.

It is standard for the scanner and parser to share the definition of token codes to guarantee that consistent values are seen by both. The file `y.tab.h`, produced by the Yacc parser generator (see Chapter Chapter:global:seven), is often used to define shared token codes. A Lex token specification consists of three sections delimited by the pair `%`. The general form of a Lex specification is shown in Figure 3.14.

In the simple example shown in Figure 3.13, we use only the second section, in which regular expressions and corresponding C code are specified. The regular expressions are simple single-character strings that match only themselves. The code executed returns a constant value representing the appropriate ac token.

We could quote the strings representing the reserved keywords (`f`, `i`, or `p`), but since these strings contain no delimiters or operators, quoting them is unnecessary. If you want to quote such strings to avoid any chance of misinterpretation, that's fine with Lex.

3.4.2 The Character Class

Our specification so far is incomplete. None of the other tokens in ac have been correctly handled, particularly identifiers and numbers. To do this, we introduce a useful concept: the **character class**. A character class is a set of characters treated identically in a token definition. Thus, in the definition of an ac identifier, all letters (except `f`, `i`, and `p`) form a class, since any of them can be used to form an identifier. Similarly in a number, any of the ten digits characters can be used.

A character class is delimited by `[` and `]`; individual characters are catenated without any quotation or separators. However `\`, `^`, `]`, and `-` must be escaped because of their special meanings in character classes. Thus `[xyz]` represents the class that can match a single `x`, `y`, or `z`. The expression `[\]]` represents the class

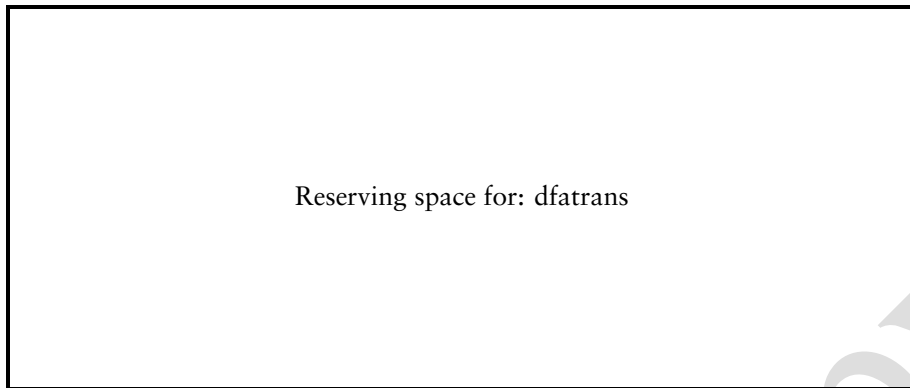


Figure 3.11: The transducer for Pascal quoted strings.

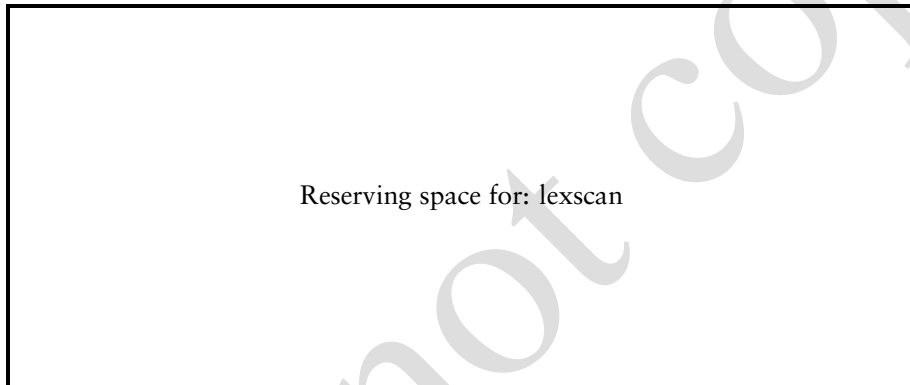


Figure 3.12: The operation of the Lex scanner generator.

```
%%  
f      { return(FLOATDCL); }  
i      { return(INTDCL); }  
p      { return(PRINT); }  
%%
```

Figure 3.13: A Lex definiton for ac's reserved words.

```
declarations  
%%  
regular expression rules  
%%  
subroutine definitions
```

Figure 3.14: The structure of Lex definiton files.

Character Class	Set of Characters Denoted
[abc]	Three characters: a, b, and c
[cba]	Three characters: a, b, and c
[a-c]	Three characters: a, b, and c
[aabbcc]	Three characters: a, b, and c
[^abc]	All characters except a, b, and c
[\^-\]]	Three characters: ^, -, and]
[^]	All characters
"[abc]"	Not a character class. This is an example of one five-character <i>string</i> : [abc].

Figure 3.15: Lex character class definitions.

```
%%
[a-eghj-oq-z]      { return(ID); }
%%
```

Figure 3.16: A Lex definition for ac's identifiers.

that can match a single] or). (The] is escaped so that it isn't misinterpreted as the end-of-character-class symbol.)

Ranges of characters are separated by a -; for example, [x-z] is the same as [xyz]. [0-9] is the set of all digits, and [a-zA-Z] is the set of all letters, both uppercase and lowercase. \ is the escape character; it is used to represent unprintables and to escape special symbols. Following C conventions, \n is the newline (that is, end-of-line), \t is the tab character, \\ is the backslash symbol itself, and \010 is the character corresponding to 10 in octal (base 8) form.

The ^ symbol complements a character class; it is Lex's representation of the Not() operation. For example, [^xy] is the character class that matches any single character except x and y. The ^ symbol applies to all characters that follow it in the character class definition, so [^0-9] is the set of all characters that aren't digits. [^] can be used to match all characters. (Avoid the use of \0 in character classes because it can be confused with the null character's special use as the end-of-string terminator in C.) Figure 3.15 illustrates various character classes and the character sets they define.

Using character classes, we can easily define ac identifiers, as shown in Figure 3.16. The character class includes the range of characters, a to e, then g and h, and then the range j to o, followed by the range q to z. We can concisely represent the 23 characters that may form an ac identifier without having to enumerate them all.


```

%%
(" ")+          { /* delete blanks */}
f               { return(FLOATDCL); }
i               { return(INTDCL); }
p               { return(PRINT); }
[a-eghj-oq-z]  { return(ID); }
([0-9]+) | ([0-9]+ "." [0-9]+)  { return(NUM); }
"="            { return(ASSIGN); }
"+"           { return(PLUS); }
"-"           { return(MINUS); }
%%

```

Figure 3.17: A Lex definition for ac's tokens.

3.4.3 Using Regular Expressions to Define Tokens

Tokens are defined using regular expressions. Lex provides the standard regular expression operators, as well as others. Catenation is specified by the juxtaposition of two expressions; no explicit operator is used. Thus `[ab][cd]` will match any of `ad`, `ac`, `bc`, or `bd`. Individual letters and numbers match themselves when outside of character class brackets. Other characters should be quoted (to avoid misinterpretation as regular expression operators.) For example, `while` (as used in C, C++, and Java) can be matched by the expressions `while`, `"while"`, or `[w][h][i][l][e]`.

Case *is* significant. The alternation operator is `|`. As usual, parentheses can be used to control grouping of subexpressions. Therefore, to match the reserved word `while` and allow any mixture of uppercase and lowercase (as required in Pascal and Ada), we can use

```
(w|W)(h|H)(i|I)(l|L)(e|E)
```

Postfix operators `*` (Kleene closure) and `+` (positive closure) are also provided, as is `?` (optional inclusion). For example, `expr?` matches `expr` zero times or once. It is equivalent to `(expr) | λ` and obviates the need for an explicit `λ` symbol. The character `.` matches any single character (other than a newline). The character `^` (when used outside a character class) matches the beginning of a line. Similarly, the character `$` matches the end of a line. Thus `^A.*e$` could be used to match an entire line that begins with `A` and ends with `e`. We now define all of ac's tokens using Lex's regular expression facilities. This is shown in Figure 3.17.

Recall that a Lex specification of a scanner consists of three sections. The first, not used so far, contains symbolic names associated with character classes and regular expressions. There is one definition per line. Each definition line contains an identifier and a definition string, separated by a blank or tab. The `{` and `}` symbols signal the macro-expansion of a symbol defined in the first section. For example, in the definition

```
Letter [a-zA-Z]
```

```

%%
Blank                " "
Digits               [0-9]+
Non_f_i_p            [a-eghj-oq-z]
%%
{Blank}+            { /* delete blanks */}
f                   { return(FLOATDCL); }
i                   { return(INTDCL); }
p                   { return(PRINT); }
{Non_f_i_p}         { return(ID); }
{Digits}|({Digits}."{Digits})
"="                 { return(ASSIGN); }
"+"                 { return(PLUS); }
"_"                 { return(MINUS); }
%%

```

Figure 3.18: An alternative definition for ac's tokens.

the expression `{Letter}` expands to `[a-zA-Z]`. Symbolic definitions can often make Lex specifications easier to read, as illustrated in Figure 3.18.

In the first section can also include source code, delimited by `%{` and `%}`, that is placed before the commands and regular expressions of section two. This source code may include statements, as well as variable, procedure, and type declarations that are needed to allow the commands of section two to be compiled. For example,

```

%{
#include "tokens.h"
%}

```

can include the definitions of token values returned when tokens are matched.

As shown earlier in the chapter, Lex's second section defines a table of regular expressions and corresponding commands in C. The first blank or tab not escaped or not part of a quoted string or character class is taken as the end of the regular expression. Thus you should avoid embedded blanks that are within regular expressions.

When an expression is matched, its associated command is executed. If an input sequence matches no expression, the sequence is simply copied verbatim to the standard output file. Input that is matched is stored in a global string variable `ytext` (whose length is `yylen`). Commands may alter `ytext` in any way. The default size of `ytext` is determined by `YYLMAX`, which is initially defined to be 200. *All* tokens, even those that will be ignored like comments, are stored in `ytext`. Hence, you may need to redefine `YYLMAX` to avoid overflow. An alternative approach to scanning comments that is not prone to the danger of overflowing `ytext` involves the use of start conditions (see [LS75] or [LMB92]).

Flex, an improved version of Lex discussed in the next section, automatically extends the size of `yytext` when necessary. This removes the danger that a very long token may overflow the text buffer.

The content of `yytext` is overwritten as each new token is scanned. Therefore you must be careful if you return the text of a token by simply returning a pointer into `yytext`. You must copy the content of `yytext` (by using perhaps `strcpy()`) *before* the next call to `yylex()`.

Lex allows regular expressions to overlap (that is, to match the same input sequences). In the case of overlap, two rules are used to determine which regular expression is matched:

1. The longest possible match is performed. Lex automatically buffers characters while deciding how many characters can be matched.
2. If two expressions match exactly the same string, the earlier expression (in order of definition in the Lex specification) is preferred.

Reserved words, for example, are often special cases of the pattern used for identifiers. So their definitions are placed before the expression that defines an identifier token. Often a “catch-all” pattern is placed at the very end of section two. It is used to catch characters that don't match any of the earlier patterns and hence are probably erroneous. Recall that “.” matches any single character (other than a newline). It is useful in a catch-all pattern. However, avoid a pattern such as `.*` because it will consume all characters up to the next newline.

3.4.4 Character Processing Using Lex

Although Lex is often used to produce scanners, it is really a general-purpose character-processing tool, programmed using regular expressions. Lex provides no character-tossing mechanism because this would be too special-purpose. So we may need to process the token text (stored in `yytext`) before returning a token code. This is normally done by calling a subroutine in the command associated with a regular expression. The definitions of such subroutines may be placed in the final section of the Lex specification. For example, we might want to call a subroutine to insert an identifier into a symbol table before it is returned to the parser. For `ac`, the line

```
{Non_f_i_p}      {insert(yytext); return(ID);}
```

could do this, with `insert` defined in the final section. Alternatively, the definition of `insert` could be placed in a separate file containing symbol table routines. This would allow `insert` to be changed and recompiled without Lex's having to be rerun. (Some implementations of Lex generate scanners rather slowly.)

In Lex, end-of-file is not handled by regular expressions. A predefined `EndFile` token, with a token code of zero, is automatically returned when end-of-file is reached at the beginning of a call to `yylex()`. It is up to the parser to recognize the zero return value as signifying the `EndFile` token.

If more than one source file must be scanned, this fact is hidden inside the scanner mechanism. `yyllex()` uses three user-defined functions to handle character-level I/O:

`input()` Reads a single character, zero on end-of-file.
`output(c)` Writes a single character to output.
`unput(c)` Puts a single character back into the input to be reread.

When `yyllex()` encounters end-of-file, it calls a user-supplied integer function named `yywrap()`. The purpose of this routine is to “wrap up” input processing. It returns the value 1 if there is no more input. Otherwise, it returns zero and arranges for `input()` to provide more characters.

The definitions for the `input()`, `output()`, `unput()`, and `yywrap()` functions may be supplied by the compiler writer (usually as C macros). Lex supplies default versions that read characters from the standard input and write them to the standard output. The default version of `yywrap()` simply returns 1, thereby signifying that there is no more input. (The use of `output()` allows Lex to be used as a tool for producing stand-alone data “filters” for transforming a stream of data.)

Lex-generated scanners normally select the longest possible input sequence that matches some token definition. Occasionally this can be a problem. For example, if we allow FORTRAN-like fixed-decimal literals such as `1.` and `.10` and the Pascal subrange operator `..`, then `1..10` will most likely be mis-scanned as two fixed-decimal literals rather than two integer literals separated by the subrange operator. Lex allows us to define a regular expression that applies only if some other expression immediately follows it. For example, `r/s` tells Lex to match regular expression `r` but only if regular expression `s` immediately follows it. `s` is *right-context*. That is, it isn't part of the token that is matched, but it must be present for `r` to be matched. Thus `[0-9]+/".."` would match an integer literal, but only if `..` immediately follows it. Since this pattern covers more characters than the one defining a fixed-decimal literal, it takes precedence. The longest match is still chosen, but the right-context characters are returned to the input so that they can be matched as part of a later token.

The operators and special symbols most commonly used in Lex are summarized in Figure 3.19. Note that a symbol sometimes has one meaning in a regular expression and an entirely different meaning in a character class (that is, within a pair of brackets). If you find Lex behaving unexpectedly, it's a good idea to check this table to be sure how the operators and symbols you've used behave. Ordinary letters and digits, as well as symbols not mentioned (such as `@`), represent themselves. If you're not sure whether a character is special, you can always escape it or make it part of a quoted string.

In summary, Lex is a very flexible generator that can produce a complete scanner from a succinct definition. The difficult part of working with Lex is learning its notation and rules. Once you've done this, Lex will relieve you of the many of chores of writing a scanner (for example, reading characters, buffering them, and deciding which token pattern matches). Moreover, Lex's notation for representing regular expressions is used in other Unix programs, most notably the `grep` pattern matching utility.

Lex can also transform input as a preprocessor, as well as scan it. It provides a number of advanced features beyond those discussed here. It does require that code segments be written in C, and hence it is not language-independent.

3.5 Other Scanner Generators

Lex is certainly the most widely known and widely available scanner generator because it is distributed as part of the Unix system. Even after years of use, it still has bugs, however, and produces scanners too slow to be used in production compilers. This section discussed briefly some of the alternatives to Lex, including Flex, JLex, Alex, Lexgen, GLA, and re2c.

It has been shown that Lex can be improved so that it is always faster than a handwritten scanner [Jac87]. This is done using *Flex*, a widely used, freely distributed Lex clone. It produces scanners that are considerably faster than the ones produced by Lex. It also provides options that allow the tuning of the scanner size versus its speed, as well as some features that Lex does not have (such as support for 8-bit characters). If Flex is available on your system, you should use it instead of Lex.

Lex also has been implemented in languages other than C. JLex [Ber97] is a Lex-like scanner generator written in Java that generates Java scanner classes. It is of particular interest to people writing compilers in Java. Alex [NF88] is an Ada version of Lex. Lexgen [AMT89] is an ML version of Lex.

An interesting alternative to Lex is GLA (Generator for Lexical Analyzers) [Gra88]. GLA takes a description of a scanner based on regular expressions and a library of common lexical idioms (such as “Pascal comment”) and produces a *directly executable* (that is, not transition table-driven scanner written in C. GLA was designed with both ease of use and efficiency of the generated scanner in mind. Experiments show it to be typically twice as fast as Flex and only slightly slower than a trivial program that reads and “touches” each character in an input file. The scanners it produces are more than competitive with the best handcoded scanners.

Another tool that produces directly executable scanners is re2c [BC93]. The scanners it produces are easily adaptable to a variety of environments and yet scanning speed is excellent.

Scanner generators are usually included as parts of complete suites of compiler development tools. These suites are often available on Windows and Macintosh systems as well as on Unix systems. Among the most widely used and highly recommended of these are DLG (part of the PCCTS tools suite, [PDC89]), CoCo/R [Moe91], an integrated scanner/parser generator, and Rex [Gro89], part of the Karlsruhe Cocktail tools suite.

Symbol	Meaning in Regular Expressions	Meaning in Character Classes
(matches with) to group subexpressions.	Represents itself.
)	matches with (to group subexpressions.	Represents itself.
[Begins a character class.	Represents itself.
]	Represents itself.	Ends a character class.
{	Matches with } to signal macro-expansion.	Represents itself.
}	Matches with { to signal macro-expansion.	Represents itself.
"	Matches with " to delimit strings.	Represents itself.
\	Escapes individual characters. Also used to specify a character by its octal code.	Escapes individual characters. Also used to specify a character by its octal code.
.	Matches any one character except \n.	Represents itself.
	Alternation (or) operator.	Represents itself.
*	Kleene closure operator (zero or more matches).	Represents itself.
+	Positive closure operator (one or more matches).	Represents itself.
?	Optional choice operator (one or more matches)	Represents itself.
/	Context sensitive matching operator.	Represents itself.
^	Matches only at the beginning of a line.	Complements the remaining characters in the class.
\$	Matches only at the end of a line.	Represents itself.
-	Represents itself.	The range of characters operator.

Figure 3.19: Meaning of operators and special symbols in Lex.

3.6 Practical Considerations of Building Scanners

In this section, we discuss the practical considerations involved in building real scanners for real programming languages. As one might expect, the finite automaton model developed earlier in the chapter sometimes falls short and must be supplemented. Efficiency concerns must be addressed. In addition, some provision for error handling must be incorporated.

We discuss a number of potential problem areas. In each case, solutions are weighed, particularly in conjunction with the Lex scanner generator discussed in Section 3.4.

3.6.1 Processing Identifiers and Literals

In simple languages that have only global variables and declarations, the scanner commonly will immediately enter an identifier into the symbol table, if it is not already there. Whether the identifier is entered or is already in the table, a pointer to the symbol table entry is then returned from the scanner.

In block-structured languages, the scanner generally is not expected to enter or look up identifiers in the symbol table because an identifier can be used in many contexts (for example, as a variable, member of a class, or label). The scanner usually cannot know when an identifier should be entered into the symbol table for the current scope or when it should return a pointer to an instance from an earlier scope. Some scanners just copy the identifier into a private string variable (that can't be overwritten) and return a pointer to it. A later compiler phase, the type checker, then resolves the identifier's intended usage.

Sometimes a **string space** is used to store identifiers in conjunction with a symbol table (see Chapter Chapter:global:eight). A string space is an extendible block of memory used to store the text of identifiers. A string space eliminates frequent calls to memory allocators such as `new` or `malloc` to allocate private space for a string. It also avoids the space overhead of storing multiple copies of the same string. The scanner can enter an identifier into the string space and return a string space pointer rather than the actual text.

An alternative to a string space is a hash table that stores identifiers and assigns to each a unique **serial number**. A serial number is a small integer that can be used instead of a string space pointer. All identifiers that have the same text get the same serial number; identifiers with different texts get different serial numbers. Serial numbers are ideal indices into symbol tables (which need not be hashed) because they are small, contiguously assigned integers. A scanner can hash an identifier when it is scanned and return its serial number as part of the identifier token.

In some languages, such as C, C++, and Java, case is significant; in others, such as Ada and Pascal, it is not. When case is significant, identifier text must be stored or returned exactly as it was scanned. Reserved word lookup must distinguish between identifiers and reserved words that differ only in case. However, when case is insignificant, case differences in the spelling of an identifier or reserved word must be guaranteed to not cause errors. An easy way to do this is to put

all tokens scanned as identifiers into a uniform case before they are returned or looked up in a reserved word table.

Other tokens, such as literals, require processing before they are returned. Integer and real (floating) literals are converted to numeric form and returned as part of the token. Numeric conversion can be tricky because of the danger of overflow or roundoff errors. It is wise to use standard library routines such as `atoi` and `atof` (in C) and `Integer.intValue` and `Float.floatValue` (in Java). For string literals, a pointer to the text of the string (with escaped characters expanded) should be returned.

The design of C contains a flaw that requires a C scanner to do a bit of special processing. Consider the character sequence `a (* b);`

This can be a call to procedure `a`, with `*b` as the parameter. If `a` has been declared in a `typedef` to be a type name, this character sequence also can be the declaration of an identifier `b` that is a pointer variable (the parentheses are not needed, but they are legal).

C contains no special marker that separates declarations from statements, so the parser will need some help in deciding whether it is seeing a procedure call or a variable declaration. One way to do this is for the scanner to create, while scanning and parsing, a table of currently visible identifiers that have been defined in `typedef` declarations. When an identifier in this table is scanned, a special `typeid` token is returned (rather than an ordinary `identifier` token). This allows the parser to distinguish the two constructs easily, since they now begin with different tokens.

Why does this complication exist in C? The `typedef` statement was not in the original definition of C in which the lexical and syntactic rules were established. When the `typedef` construct was added, the ambiguity was not immediately recognized (parentheses, after all, are rarely used in variable declarations). When the problem was finally recognized, it was too late, and the “trick” described previously had to be devised to resolve the correct usage.

Processing Reserved Words

Virtually all programming languages have symbols (such as `if` and `while`) that match the lexical syntax of ordinary identifiers. These symbols are called *keywords*. If the language has a rule that keywords may not be used as programmer-defined identifiers, then they are *reserved words*, that is, they are reserved for special use.

Most programming languages choose to make keywords reserved. This simplifies parsing, which drives the compilation process. It also makes programs more readable. For example, in Pascal and Ada, subprograms without parameters are called as `name`; (no parentheses required). But what if, for example, `begin` and `end` are not reserved and some devious programmer has declared procedures named `begin` and `end`? The result is a program whose meaning is not well-defined, as shown in the following example, which can be parsed in many ways:


```

begin
  begin;
  end;
  end;
  begin;
end

```

With careful design, you can avoid outright ambiguities. For example, in PL/I keywords are not reserved; procedures are called using an explicit `call` keyword. Nonetheless, opportunities for convoluted usage abound. Keywords may be used as variable names, allowing the following:

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

The problem with reserved words is that if they are too numerous, they may confuse inexperienced programmers, who may unknowingly choose an identifier name that clashes with a reserved word. This usually causes a syntax error in a program that “looks right” and in fact *would* be right if the symbol in question was not reserved. COBOL is infamous for this problem because it has several hundred reserved words. For example, in COBOL, `zero` is a reserved word. So is `zeros`. So is `zeroes`!

In Section 3.4.1, we showed how to recognize reserved words by creating distinct regular expressions for each. This approach was feasible because Lex (and Flex) allows more than one regular expression to match a character sequence, with the earliest expression that matches taking precedence. Creating regular expressions for each reserved word increases the number of states in the transition table that a scanner generator creates. In as simple a language as Pascal (which has only 35 reserved words), the number of states increases from 37 to 165 [Gra88]. With the transition table in uncompressed form and having 127 columns for ASCII characters (excluding null), the number of transition table entries increases from 4,699 to 20,955. This may not be a problem with modern multimegabyte memories. Still, some scanner generators, such as Flex, allow you to choose to optimize scanner size or scanner speed.

Exercise 18 establishes that any regular expression may be complemented to obtain all strings not in the original regular expression. That is, \bar{A} , the complement of A , is regular if A is. Using complementation of regular expressions, we can write a regular expression for nonreserved identifiers:

(*ident* | *if* | *while* | ...)

That is, if we take the complement of the set containing reserved words and all nonidentifier strings, we get all strings that *are* identifiers, *excluding* the reserved words. Unfortunately, neither Lex nor Flex provides a complement operator for regular expressions (\sim works only on character sets).

We could just write down a regular expression directly, but this is too complex to consider seriously. Suppose `END` is the only reserved word and identifiers contain only letters. Then

$$L \mid (LL) \mid ((LLL)L^+) \mid ((L - ' E')L^*) \mid (L(L - ' N')L^*) \mid (LL(L - ' D')L^*)$$

defines identifiers that are shorter or longer than three letters, that do not start with E, that are without N in position two, and so on.

Many hand-coded scanners treat reserved words as ordinary identifiers (as far as matching tokens is concerned) and then use a separate table lookup to detect them. Automatically generated scanners can also use this approach, especially if transition table size is an issue. After an apparent identifier is scanned, an exception table is consulted to see if a reserved word has been matched. When case is significant in reserved words, the exception lookup requires an exact match. Otherwise, the token should be translated to a standard form (all uppercase or lowercase) before the lookup.

An exception table may have any of various organizations. An obvious one is a sorted list of exceptions suitable for a binary search. A hash table also may be used. For example, the length of a token may be used as an index into a list of exceptions of the same length. If exception lengths are well-distributed, few comparisons will be needed to determine whether a token is an identifier or a reserved word. [Cic86] showed that perfect hash functions are possible. That is, each reserved word is mapped to a unique position in the exception table and no position in the table is unused. A token is either the reserved word selected by the hash function or an ordinary identifier.

If identifiers are entered into a string space or given a unique serial number by the scanner, then reserved words can be entered in advance. Then when what looks like an identifier is found to have a serial number or string space position *smaller* than the initial position assigned to identifiers, we know that a reserved word rather than an identifier has been scanned. In fact, with a little care we can assign initial serial numbers so that they match exactly the token codes used for reserved words. That is, if an identifier is found to have a serial number s , where s is less than the number of reserved words, then s must be the correct token code for the reserved word just scanned.

3.6.2 Using Compiler Directives and Listing Source Lines

Compiler directives and pragmas control compiler options (for example, listings, source file inclusion, conditional compilation, optimizations, and profiling). They may be processed either by the scanner or by subsequent compiler phases. If the directive is a simple flag, it can be extracted from a token. The command is then executed, and finally the token is deleted. More elaborate directives, such as Ada pragmas, have nontrivial structure and need to be parsed and translated like any other statement.

A scanner may have to handle source inclusion directives. These directives cause the scanner to suspend the reading of the current file and begin the reading and scanning of the contents of the specified file. Since an included file may itself contain an include directive, the scanner maintains a stack of open files. When the file at the top of the stack is completely scanned, it is popped and scanning resumes with the file now at the top of the stack. When the entire stack is empty, end-of-file is recognized and scanning is completed. Because C has a rather elaborate macro definition and expansion facility, macro processing and

included files are typically handled by a preprocessing phase prior to scanning and parsing. The preprocessor, `cpp`, may in fact be used with languages other than C to obtain the effects of source file inclusion, macro processing, and so on.

Some languages (such as C and PL/I) include conditional compilation directives that control whether statements are compiled or ignored. Such directives are useful in creating multiple versions of a program from a common source. Usually, these directives have the general form of an `if` statement; hence, a conditional expression will be evaluated. Characters following the expression will then either be scanned and passed to the parser or be ignored until an `end if` delimiter is reached. If conditional compilation structures can be nested, a skeletal parser for the directives may be needed.

Another function of the scanner is to list source lines and to prepare for the possible generation of error messages. While straightforward, this requires a bit of care. The most obvious way to produce a source listing is to echo characters as they are read, using end-of-line characters to terminate a line, increment line counters, and so on. This approach has a number of shortcomings, however.

- Error messages may need to be printed. These should appear merged with source lines, with pointers to the offending symbol.
- A source line may need to be edited before it is written. This may involve inserting or deleting symbols (for example, for error repair), replacing symbols (because of macro preprocessing), and reformatting symbols (to prettyprint a program, that is, to print a program with text properly indented, `if-else` pairs aligned, and so on).
- Source lines that are read are not always in a one-to-one correspondence with source listing lines that are written. For example, in Unix a source program can legally be condensed into a single line (Unix places no limit on line lengths). A scanner that attempts to buffer entire source lines may well overflow buffer lengths.

In light of these considerations, it is best to build output lines (which normally are bounded by device limits) *incrementally* as tokens are scanned. The token image placed in the output buffer may not be an exact image of the token that was scanned, depending on error repair, prettyprinting, case conversion, or whatever else is required. If a token cannot fit on an output line, the line is written and the buffer is cleared. (To simplify editing, you should place source line numbers in the program's listing.) In rare cases, a token may need to be broken; for example, if a string is so long that its text exceeds the output line length.

Even if a source listing is not requested, each token should contain the line number in which it appeared. The token's position in the source line may also be useful. If an error involving the token is noted, the line number and position marker can be used to improve the quality of error messages. By specifying where in the source file the error occurred. It is straightforward to open the source file and then list the source line containing the error, with the error message immediately below it. Sometimes, an error may not be detected until long after

the line containing the error has been processed. An example of this is a `goto` to an undefined label. If such delayed errors are rare (as they usually are), a message citing a line number can be produced, for example, “Undefined label in statement 101.” In languages that freely allow forward references, delayed errors may be numerous. For example, Java allows declarations of methods after they are called. In this case, a file of error messages keyed with line numbers can be written and later merged with the processed source lines to produce a complete source listing. Source line numbers are also required for reporting post-scanning errors in multipass compilers. For example, a type conversion error may arise during semantic analysis; associating a line number with the error message greatly helps a programmer understand and correct the error.

A common view is that compilers should just concentrate on translation and code generation and leave the listing and prettyprinting (but not error messages) to other tools. This considerably simplifies the scanner.

3.6.3 Terminating the Scanner

A scanner is designed to read input characters and partition them into tokens. When the end of the input file is reached, it is convenient to create an end-of-file pseudo-character.

In Java, for example, `InputStream.read()`, which reads a single byte, returns `-1` when end-of-file is reached. A constant, `Eof`, defined as `-1`, can be treated as an “extended” ASCII character. This character then allows the definition of an `EndFile` token that can be passed back to the parser. The `EndFile` token is useful in a CFG because it allows the parser to verify that the logical end of a program corresponds to its physical end. In fact, LL(1) parsers (discussed in Chapter Chapter:global:five) and LALR(1) parsers (discussed in Chapter Chapter:global:six) require an `EndFile` token.

What will happen if a scanner is called after end-of-file is reached? Obviously, a fatal error could be registered, but this would destroy our simple model in which the scanner always returns a token. A better approach is to continue to return the `EndFile` token to the parser. This allows the parser to handle termination cleanly, especially since the `EndFile` token is normally syntactically valid only after a complete program is parsed. If the `EndFile` token appears too soon or too late, the parser can perform error repair or issue a suitable error message.

3.6.4 Multicharacter Lookahead

We can generalize FAs to look ahead beyond the next input character. This feature is important for implementing a scanner for FORTRAN. In FORTRAN, the statement `D0 10 J = 1,100` specifies a loop, with index `J` ranging from 1 to 100. In contrast, the statement `D0 10 J = 1.100` is an assignment to the variable `D010J`. In FORTRAN, blanks are not significant except in strings. A FORTRAN scanner can determine whether the `0` is the last character of a `D0` token only after reading as far as the comma (or period). (In fact, the erroneous substitution of a “.” for a “,” in a FORTRAN `D0` loop once caused a 1960s-era space launch to fail!

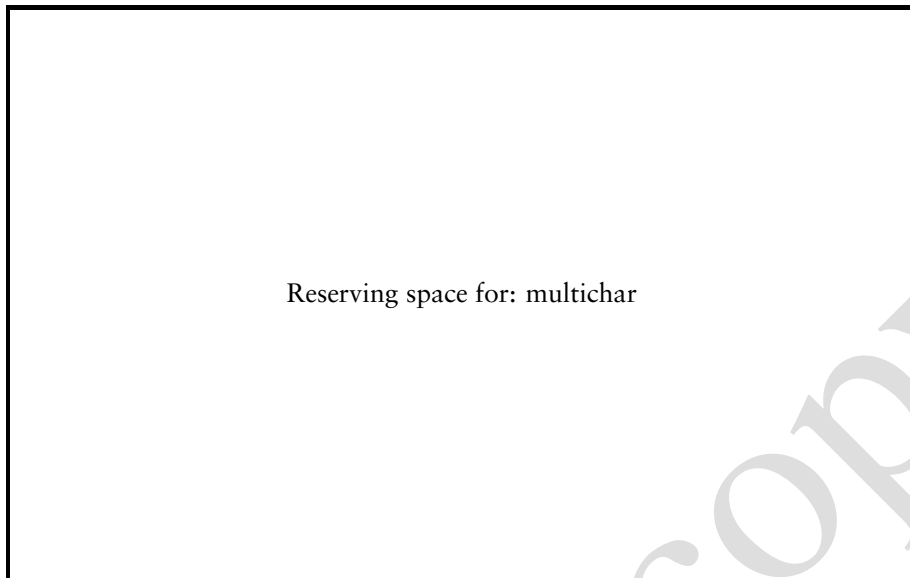


Figure 3.20: An FA that scans integer and real literals and the subrange operator.

Because the substitution resulted in a valid statement, the error was not detected until run-time, which in this case was *after* the rocket had been launched. The rocket deviated from course and had to be destroyed.)

We've already shown you a milder form of the extended lookahead problem that occurs in Pascal and Ada. Scanning, for example, `10..100` requires two-character lookahead after the `10`. Using the FA of Figure 3.20 and given `10..100`, we would scan three characters and stop in a nonaccepting state. Whenever we stop reading in a nonaccepting state, we can back up over accepted characters until an accepting state is found. Characters we back up over are rescanned to form later tokens. If no accepting state is reached during backup, we have a lexical error and invoke lexical error recovery.

In Pascal or Ada, more than two-character lookahead is not needed; this simplifies the buffering of characters to be rescanned. Alternatively, we can add a new accepting state to the previous FA that corresponds to a pseudotoken of the form $(D^+.)$. If this token is recognized, we strip the trailing “.” from the token text and buffer it for later reuse. We then return the token code of an integer literal. In effect, we are simulating the effect of a context-sensitive match as provided by Lex's `/` operator.

Multiple character lookahead may also be a consideration in scanning *invalid* programs. For example, in C (and many other programming languages) `12.3e+q` is an invalid token. Many C compilers simply flag the entire character sequence as invalid (a floating-point value with an illegal exponent). If we follow our general scanning philosophy of matching the longest *valid* character sequence, the scanner could be backed up to produce four tokens. Since this token sequence

Buffered Token	Token Flag
1	Integer literal.
12	Integer literal.
12.	Floating-point literal.
12.3	Floating-point literal.
12.3e	Invalid (but valid prefix).
12.3e+	Invalid (but valid prefix).

Figure 3.21: Building the token buffer and setting token flags when scanning with a backup.

(12.3, e, +, q) is invalid, the parser will detect a syntax error when it processes the sequence. Whether we decide to consider this a lexical error or a syntax error (or both) is unimportant. Some phase of the compiler must detect the error.

It is not difficult to build a scanner that can perform general backup. This allows the scanner to operate correctly no matter how token definitions overlap. As each character is scanned, it is buffered and a flag is set indicating whether the character sequence scanned so far is a valid token (the flag might be the appropriate token code). If we are not in an accepting state and cannot scan any more characters, backup is invoked. We extract characters from the right end of the buffer and queue them for rescanning. This process continues until we reach a prefix of the scanned characters flagged as a valid token. This token is returned by the scanner. If no prefix is flagged as valid, we have a lexical error. (Lexical errors are discussed in Section 3.6.6.)

Buffering and backup are essential in general-purpose scanners such as those generated by Lex. It is impossible to know in advance which regular expression pattern will be matched. Instead, the generated scanner (using its internal DFA) follows all patterns that are possible matches. If a particular pattern is found to be unmatchable, an alternative pattern that matches a shorter input sequence may be chosen. The scanner will back up to the longest input prefix that can be matched, saving buffered characters that will be matched in a later call to the scanner.

As an example of scanning with backup, consider the previous example of 12.3e+q. Figure 3.21 shows how the buffer is built and flags are set. When the q is scanned, backup is invoked. The longest character sequence that is a valid token is 12.3, so a floating-point literal is returned. e+ is requeued so that it can be later rescanned.

3.6.5 Performance Considerations

Our main concern in this chapter is showing how to write correct and robust scanners. Because scanners do so much character-level processing, they can be a real performance bottleneck in production compilers. Hence, it is a good idea to consider how to increase scanning speed.

```
System.out.println("Four score | and seven years ago,");
```

Figure 3.22: An example of double buffering.

One approach to increasing scanner speed is to use a scanner generator such as Flex or GLA that is designed to generate fast scanners. These generators will incorporate many “tricks” that increase speed in nonobvious ways.

If you hand-code a scanner, a few general principles can increase scanner performance dramatically.

Try to block character-level operations whenever possible. It is usually better to do one operation on n characters rather than n operations on single characters. This is most apparent in reading characters. In the examples herein, characters are input one at a time, perhaps using Java's `InputStream.read` (or a C or C++ equivalent). Using single-character processing can be quite inefficient. A subroutine call can cost hundreds or thousands of instructions to execute—far too many for a single character. Routines such as `InputStream.read(buffer)` perform block reads, putting an entire block of characters directly into `buffer`. Usually, the number of characters read is set to the size of a disk block (512 or perhaps 1,024 bytes) so that an entire disk block can be read in one operation. If fewer than the requested number of characters are returned, we know we have reached end-of-file. An end-of-file (EOF) character can be set to indicate this.

One problem with reading blocks of characters is that the end of a block won't usually correspond to the end of a token. For example, near the end of a block may be found the beginning of a quoted string but not its end. Another read operation to get the rest of the string may overwrite the first part.

Double-buffering can avoid this problem, as shown in Figure 3.22. Input is first read into the left buffer and then into the right buffer, and then the left buffer is overwritten. Unless a token whose text we want to save is longer than the buffer length, tokens can cross a buffer boundary without difficulty. If the buffer size is made large enough (say 512 or 1,024 characters), the chance of losing part of a token is very low. If a token's length is near the buffer's length, we can extend the buffer size, perhaps by using Java-style `Vector` objects rather than arrays to implement buffers.

We can speed up a scanner not only by doing block reads, but also by avoiding unnecessary copying of characters. Because so many characters are scanned, moving them from one place to another can be costly. A block read enables direct reading into the scanning buffer rather than into an intermediate input buffer. As characters are scanned, we need not copy characters from the input buffer unless we recognize a token whose text must be saved or processed (an identifier or a literal). With care, we can process the token's text directly from the input buffer.

At some point, using a profiling tool such as `qpt`, `prof`, `gprof`, or `pixie` may allow you to find unexpected performance bottlenecks in a scanner.

3.6.6 Lexical Error Recovery

A character sequence that cannot be scanned into any valid token results in a **lexical error**. Although uncommon, such errors must be handled by a scanner. It is unreasonable to stop compilation because of what is often a minor error, so usually we try some sort of *lexical error recovery*. Two approaches come to mind:

1. Delete the characters read so far and restart scanning at the next unread character.
2. Delete the first character read by the scanner and resume scanning at the character following it.

Both approaches are reasonable. The former is easy to do. We just reset the scanner and begin scanning anew. The latter is a bit harder to do but also is a bit safer (because fewer characters are immediately deleted). Rescanning non-deleted characters can be implemented using the buffering mechanism described previously for scanner backup.

In most cases, a lexical error is caused by the appearance of some illegal character, which usually appears as the beginning of a token. In this case, the two approaches work equally well. The effects of lexical error recovery might well create a syntax error, which will be detected and handled by the parser. Consider `...for$tnight...`. The `$` would terminate scanning of `for`. Since no valid token begins with `$`, it would be deleted. Then `tnight` would be scanned as an identifier. The result would be `...for tnight...`, which will cause a syntax error. Such occurrences are unavoidable.

However, a good syntactic error-repair algorithm will often make some reasonable repair. In this case, returning a special warning token when a lexical error occurs can be useful. The semantic value of the warning token is the character string that is deleted to restart scanning. The warning token warns the parser that the next token is unreliable and that error repair may be required. The text that was deleted may be helpful in choosing the most appropriate repair.

Certain lexical errors require special care. In particular, runaway strings and comments should receive special error messages.

Handling Runaway Strings and Comments Using Error Tokens

In Java, strings are not allowed to cross line boundaries, so a runaway string is detected when an end-of-line character is reached within the string body. Ordinary recovery heuristics are often inappropriate for this error. In particular, deleting the first character (the double quote character) and restarting scanning will almost certainly lead to a cascade of further “false” errors because the string text is inappropriately scanned as ordinary input.

One way to catch runaway strings is to introduce an **error token**. An error token is not a valid token; it is never returned to the parser. Rather, it is a pattern

for an error condition that needs special handling. We use an error token to represent a string terminated by an Eol rather than a double quote. For a valid string, in which internal double quotes and backslashes are escaped (and no other escaped characters are allowed), we can use

```
" (Not( " | Eol | \) | \" | \\ )* "
```

For a runaway string, we can use

```
" (Not( " | Eol | \) | \" | \\ )* Eol
```

When a runaway string token is recognized, a special error message should be issued. Further, the string may be repaired and made into a correct string by returning an ordinary string token with the opening double quote and closing Eol stripped (just as ordinary opening and closing double quotes are stripped). Note, however, that this repair may or may not be “correct.” If the closing double quote is truly missing, the repair will be good. If it is present on a succeeding line, however, a cascade of inappropriate lexical and syntactic errors will follow until the closing double quote is finally reached.

Some PL/I compilers issue special warnings if comment delimiters appear within a string. Although such strings are legal, they almost always result from errors that cause a string to extend farther than was intended. A special string token can be used to implement such warnings. A valid string token is returned *and* an appropriate warning message is issued.

In languages such as C, C++, Java, and Pascal, which allow multiline comments, improperly terminated (that is, runaway) comments present a similar problem. A runaway comment is not detected until the scanner finds a close comment symbol (possibly belonging to some other comment) or until end-of-file is reached. Clearly, a special error message is required.

Consider the Pascal-style comments that begin with a { and end with a }. (Comments that begin and end with a pair of characters, such as /* and */ in Java, C, and C++, are a bit trickier to get right; see Exercise 6.)

Correct Pascal comments are defined quite simply: { Not()}* }

To handle comments terminated by Eof, the error token approach can be used: { Not()}* Eof

To handle comments closed by a close comment belonging to another comment (for example, {...missing close comment...{ normal comment }}, we issue a warning (but not an error message; this form of comment is lexically legal). In particular, a comment containing an open comment symbol in its body is most probably a symptom of the kind of omission depicted previously. We therefore split the legal comment definition into two tokens. The one that accepts an open comment in its body causes a warning message to be printed (“Possible unclosed comment”). The result is three token definitions:

{ Not({ | })* } and { (Not({ | })* { Not({ | })* }+) } and { Not()}* Eof

The first definition matches correct comments that do not contain an open comment in their bodies. The second matches correct, but suspect, comments

that contain at least one open comment in their bodies. The final definition is an error token that matches a “runaway comment” terminated by end-of-file.

Single-line comments, found in Java and C++, are always terminated by an end-of-line character and so do not fall prey to the runaway comment problem. They do, however, require that each line of a multiline comment contain an open comment marker. Note, too, that we mentioned previously that balanced brackets cannot be correctly scanned using regular expressions and finite automata. A consequence of this limitation is that nested comments cannot be properly scanned using conventional techniques. This limitation causes problems when we want comments to nest, particularly when we “comment-out” a piece of code (which itself may well contain comments). Conditional compilation constructs, such as `#if` and `#endif` in C and C++, are designed to safely disable the compilation of selected parts of a program.

3.7 Regular Expressions and Finite Automata

Regular expressions are equivalent to FAs. In fact, the main job of a scanner generator program such as Lex is to transform a regular expression definition into an equivalent FA. It does this by first transforming the regular expression into a **nondeterministic finite automaton** (NFA). An NFA is a generalization of a DFA that allows transitions labeled with λ as well as multiple transitions from a state that have the same label.

After creating an NFA, a scanner generator then transforms the NFA into a DFA, introduced earlier in the chapter. Exactly how it does both of these steps is discussed a little later in this section.

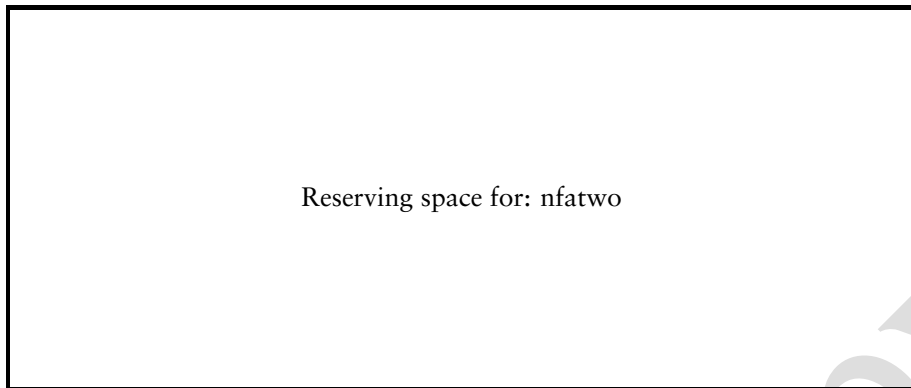
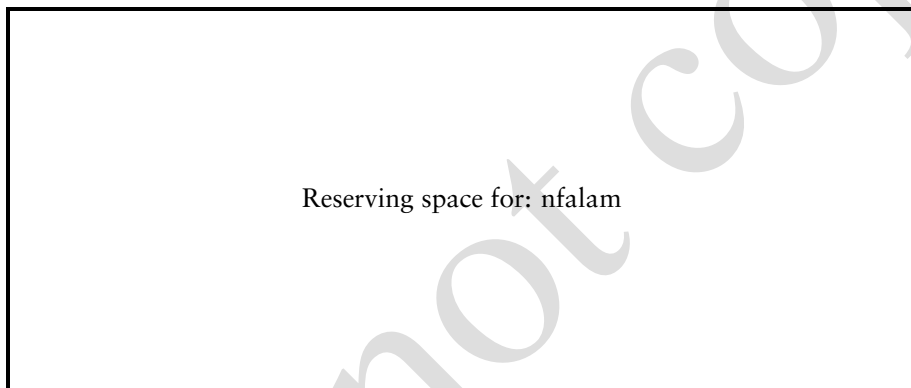
An NFA, upon reading a particular input, need not make a unique (deterministic) choice of which state to visit. For example, as shown in Figure 3.23, an NFA is allowed to have a state that has two transitions (shown by the arrows) coming out of it, labeled by the same symbol. As shown in Figure 3.24, an NFA may also have transitions labeled with λ .

Transitions are normally labeled with individual characters in Σ , and although λ is a string (the string with no characters in it), it is definitely *not* a character. In the last example, when the FA is in the state at the left and the next input character is a , it may choose either to use the transition labeled a or to first follow the λ transition (you can always find λ wherever you look for it) and *then* follow an a transition. FAs that contain no λ transitions and that always have unique successor states for any symbol are *deterministic*.

The algorithm to make an FA from a regular expression proceeds in two steps. First, it transforms the regular expression into an NFA. Then it transforms the NFA into a DFA.

3.7.1 Transforming a Regular Expression into an NFA

Transforming a regular expression into an NFA is easy. A regular expression is built of the *atomic* regular expressions a (where a is a character in Σ) and λ by

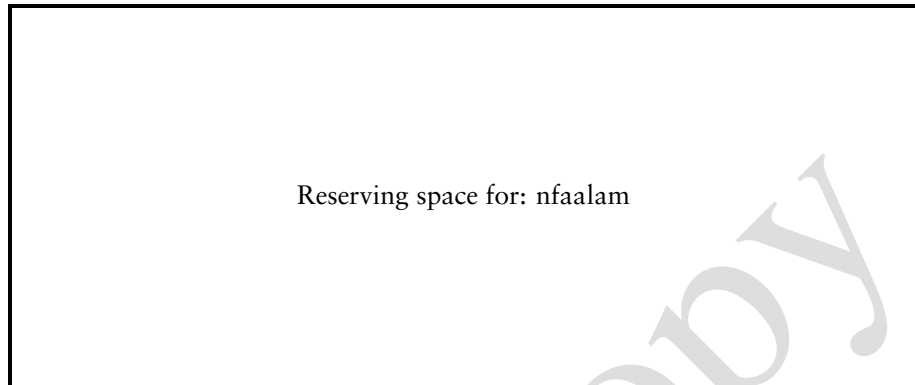
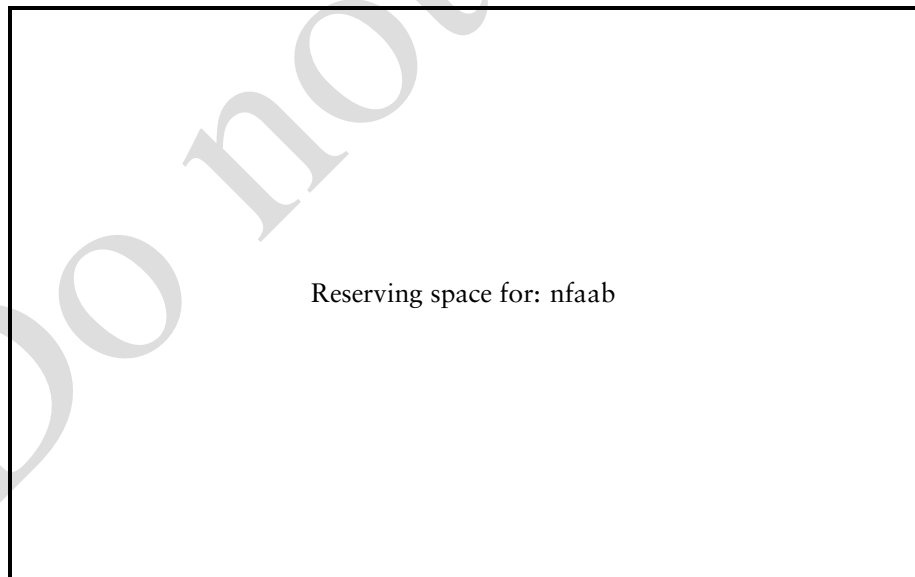
Figure 3.23: An NFA with two a transitions.Figure 3.24: An NFA with a λ transition.

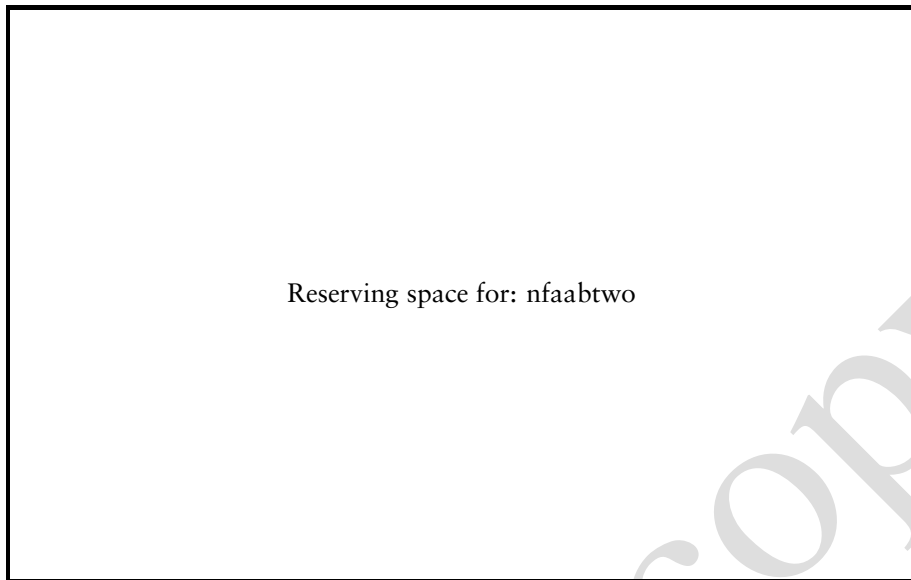
using the three operations AB , $A \mid B$, and A^* . Other operations (such as A^+) are just abbreviations for combinations of these. As shown in Figure 3.25, NFAs for a and λ are trivial.

Now suppose we have NFAs for A and B and want one for $A \mid B$. We construct the NFA shown in Figure 3.26. The states labeled A and B were the accepting states of the automata for A and B ; we create a new accepting state for the combined FA.

As shown in Figure 3.27, the construction of AB is even easier. The accepting state of the combined FA is the same as the accepting state of B .

Finally, the NFA for A^* is shown in Figure 3.28. The start state is an accepting state, so λ is accepted. Alternatively, we can follow a path through the FA for A one or more times so that zero or more strings that belong to A are matched.

Figure 3.25: NFAs for a and λ .Figure 3.26: An NFA for $A \mid B$.

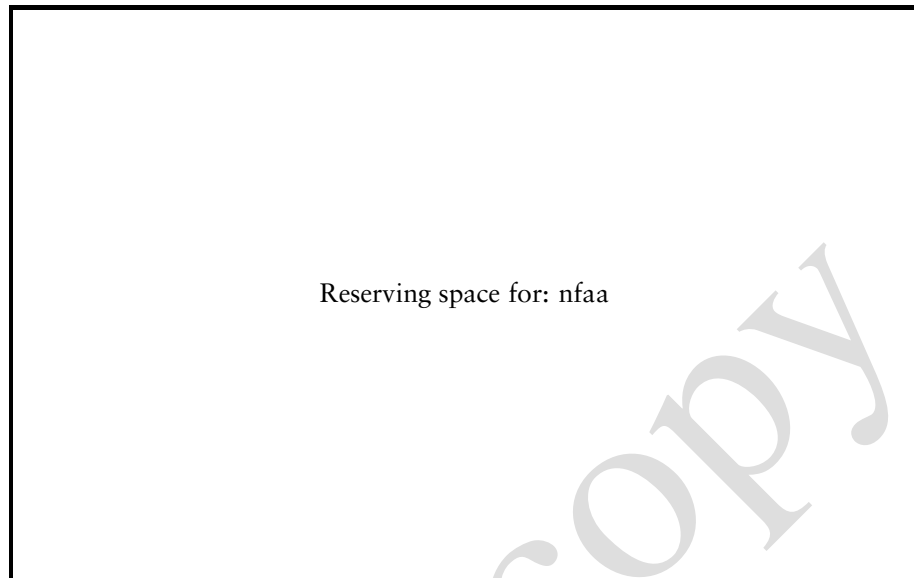
Figure 3.27: An NFA for AB .

3.7.2 Creating the DFA

The transformation from an NFA N to an equivalent DFA D works by what is sometimes called the **subset construction**. The subset construction algorithm is shown in Figure 3.29. The algorithm associates each state of D with a *set* of states of N . The idea is that D will be in state $\{x, y, z\}$ after reading a given input string if, and only if, N could be in *any* of the states x , y , or z , depending on the transitions it chooses. Thus D keeps track of all of the possible routes N might take and runs them simultaneously. Because N is a *finite* automaton, it has only a finite number of states. The number of subsets of N 's states is also finite. This makes tracking various sets of states feasible.

The start state of D is the set of all states that N could be in without reading any input characters—that is, the set of states reachable from the start state of N following only λ transitions. Algorithm CLOSE, called from RECORDSTATE, in Figure 3.29 computes those states that can be reached after only λ transitions. Once the start state of D is built, we begin to create successor states.

To do this, we place each state S of D on a work list when it is created. For each state S on the work list and each character c in the vocabulary, we compute S 's successor under c . S is identified with some set of N 's states $\{n1, n2, \dots\}$. We find all of the possible successor states to $\{n1, n2, \dots\}$ under c and obtain a set $\{m1, m2, \dots\}$. Finally, we include the λ -successors of $\{m1, m2, \dots\}$. The resulting set of NFA states is included as a state in D , and a transition from S to it, labeled with c , is added to D . We continue adding states and transitions to D until all possible successors to existing states are added. Because each state

Figure 3.28: An NFA for A^* .

corresponds to a (finite) subset of N 's states, the process of adding new states to D must eventually terminate.

An accepting state of D is any set that contains an accepting state of N . This reflects the convention that N accepts if there is *any* way it could get to its accepting state by choosing the “right” transitions.

To see how the subset construction operates, consider the NFA shown in Figure 3.30. In the NFA in the figure, we start with state 1, the start state of N , and add state 2, its λ -successor. Hence, D 's start state is $\{1, 2\}$. Under a , $\{1, 2\}$'s successor is $\{3, 4, 5\}$. State 1 has itself as a successor under b . When state 1's λ -successor, 2, is included, $\{1, 2\}$'s successor is $\{1, 2\}$. $\{3, 4, 5\}$'s successors under a and b are $\{5\}$ and $\{4, 5\}$. $\{4, 5\}$'s successor under b is $\{5\}$. Accepting states of D are those state sets that contain N 's accepting state (5). The resulting DFA is shown in Figure 3.31.

It can be established that the DFA constructed by MAKEDETERMINISTIC is equivalent to the original NFA (see Exercise 20). What is not obvious is the fact that the DFA that is built can sometimes be *much* larger than the original NFA. States of the DFA are identified with sets of NFA states. If the NFA has n states, there are 2^n distinct sets of NFA states and hence the DFA may have as many as 2^n states. Exercise 16 discusses an NFA that actually exhibits this exponential blowup in size when it is made deterministic. Fortunately, the NFAs built from the kind of regular expressions used to specify programming language tokens do not exhibit this problem when they are made deterministic. As a rule, DFAs used for scanning are simple and compact.

When creating a DFA is impractical (either because of speed-of-generation

```

function MAKEDETERMINISTIC( $N$ ) : DFA
   $D.StartState \leftarrow RECORDSTATE(\{N.StartState\})$ 
  foreach  $S \in WorkList$  do
     $WorkList \leftarrow WorkList - \{S\}$ 
    foreach  $c \in \Sigma$  do  $D.T(S, c) \leftarrow RECORDSTATE(\bigcup_{s \in S} N.T(s, c))$ 
   $D.AcceptStates \leftarrow \{S \in D.States \mid S \cap N.AcceptStates \neq \emptyset\}$ 
end
function CLOSE( $S, T$ ) : Set
   $ans \leftarrow S$ 
  repeat
     $changed \leftarrow false$ 
    foreach  $s \in ans$  do
      foreach  $t \in T(s, \lambda)$  do
        if  $t \notin ans$ 
        then
           $ans \leftarrow ans \cup \{t\}$ 
           $changed \leftarrow true$ 
        end
      end
    until not  $changed$ 
  return ( $ans$ )
end
function RECORDSTATE( $s$ ) : Set
   $s \leftarrow CLOSE(s, N.T)$ 
  if  $s \notin D.States$ 
  then
     $D.States \leftarrow D.States \cup \{s\}$ 
     $WorkList \leftarrow WorkList \cup \{s\}$ 
  end
  return ( $s$ )
end

```

Figure 3.29: Construction of a DFA D from an NFA N .

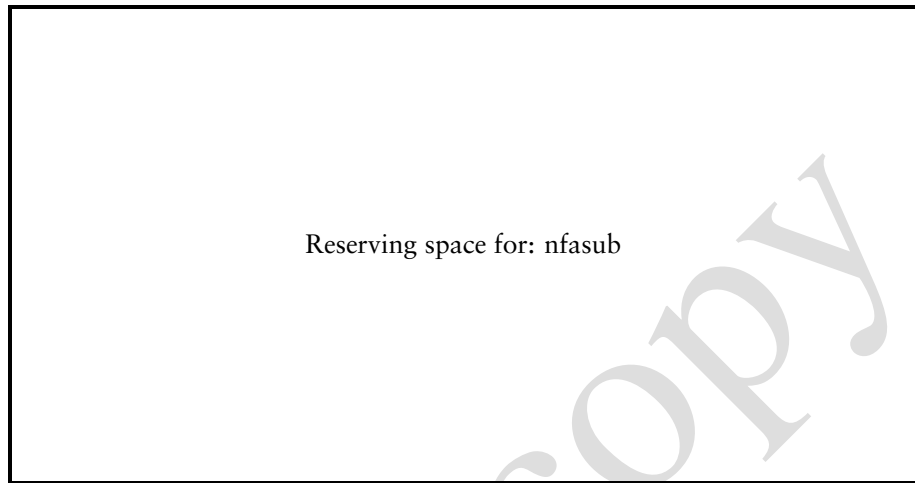


Figure 3.30: An NFA showing how subset construction operates.

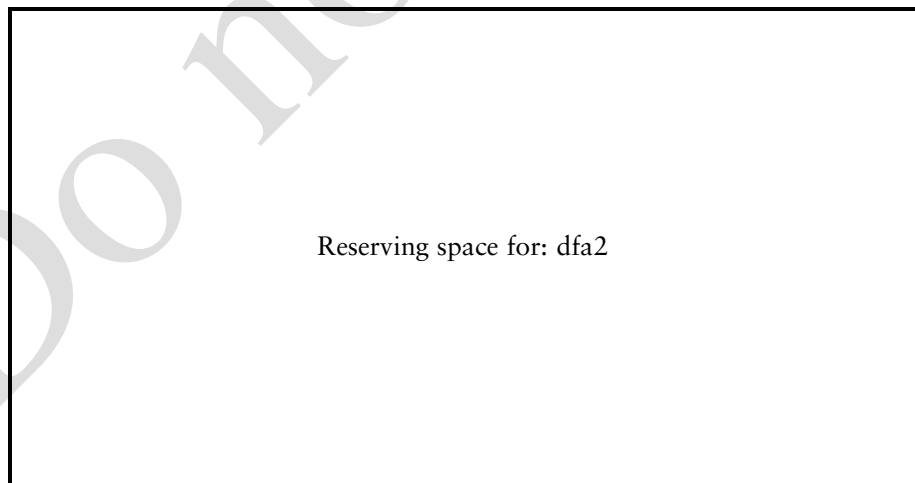


Figure 3.31: DFA created for NFA of Figure 3.30.

or size concerns), an alternative is to scan using an NFA (see Exercise 17). Each possible path through an NFA can be tracked, and reachable accepting states can be identified. Scanning is slower using this approach, so it is usually used only when the construction of a DFA is not cost-effective.

3.7.3 Optimizing Finite Automata

We can improve the DFA created by MAKEDETERMINISTIC. Sometimes this DFA will have more states than necessary. For every DFA, there is a *unique* smallest (in terms of number of states) equivalent DFA. Suppose a DFA D has 75 states and there is a DFA D' with 50 states that accepts exactly the same set of strings. Suppose further that no DFA with fewer than 50 states is equivalent to D . Then D' is the only DFA with 50 states equivalent to D . Using the techniques discussed in this section, we can optimize D by replacing it with D' .

Some DFAs contain **unreachable states**, states that cannot be reached from the start state. Other DFAs may contain **dead states**, states that cannot reach any accepting state. It is clear that neither unreachable states nor dead states can participate in scanning any valid token. So we eliminate all such states as part of our optimization process.

We optimize the resulting DFA by merging states we know to be equivalent. For example, two accepting states that have no transitions out of them are equivalent. Why? Because they behave exactly the same way—they accept the string read so far but will accept no additional characters. If two states, s_1 and s_2 , are equivalent, then all transitions to s_2 can be replaced with transitions to s_1 . In effect, the two states are merged into one common state.

How do we decide what states to merge? We take a *greedy* approach and try the most optimistic merger. By definition, accepting and nonaccepting states are distinct, so we initially try to create only two states: one representing the merger of all accepting states and the other representing the merger of all nonaccepting states. Only two states is almost certainly too optimistic. In particular, all of the constituents of a merged state must agree on the same transition for each possible character. That is, for character c all of the merged states either must have no successor under c or must go to a single (possibly merged) state. If all constituents of a merged state do not agree on the transition to follow for some character, the merged state is split into two or more smaller states that *do* agree.

As an example, assume we start with the FA shown in Figure 3.32. Initially, we have a merged nonaccepting state $\{1, 2, 3, 5, 6\}$ and a merged accepting state $\{4, 7\}$. A merger is legal if, and only if, all constituent states agree on the same successor state for all characters. For example, states 3 and 6 would go to an accepting state when given character c ; states 1, 2, and 5 would not, so a split must occur. We add an error state s_E to the original DFA that will be the successor state under any illegal character. (Thus reaching s_E becomes equivalent to detecting an illegal token.) s_E is not a real state. Rather, it allows us to assume that every state has a successor under every character. s_E is never merged with any real state.

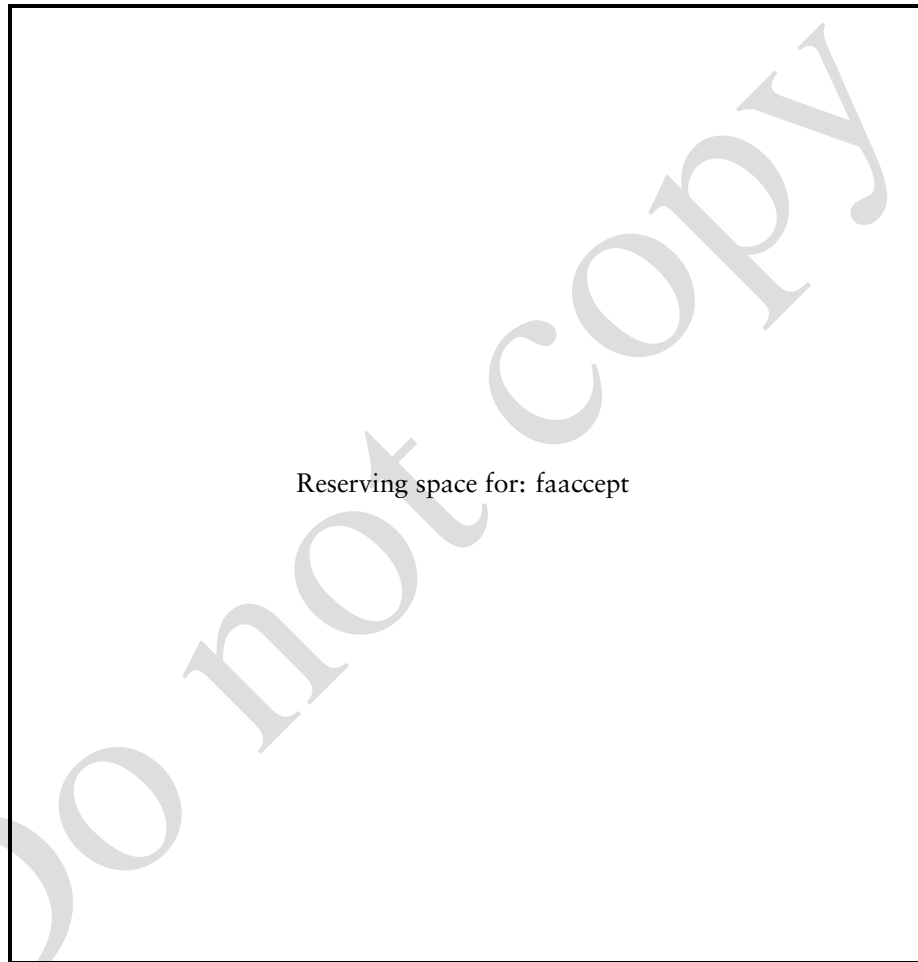


Figure 3.32: Example FA before merging.

```

procedure SPLIT(MergedStates)
  repeat
    changed  $\leftarrow$  false
    foreach  $S \in$  MergedStates,  $c \in \Sigma$  do
      targets  $\leftarrow \bigcup_{s \in S} \text{TARGETBLOCK}(s, c, \textit{MergedStates})$ 
      if  $|\textit{targets}| > 1$ 
      then
        changed  $\leftarrow$  true
        foreach  $t \in \textit{targets}$  do
          newblock  $\leftarrow \{s \in S \mid \text{TARGETBLOCK}(s, c, \textit{MergedStates}) = t\}$ 
          MergedStates  $\leftarrow \textit{MergedStates} \cup \{\textit{newblock}\}$ 
          MergedStates  $\leftarrow \textit{MergedStates} - \{S\}$ 
    until not changed
  end
function TARGETBLOCK( $s, c, \textit{MergedStates}$ ) : MergedState
  return  $\{B \in \textit{MergedStates} \mid T(s, c) \in B\}$ 
end

```

Figure 3.33: An algorithm to split FA states.

Algorithm SPLIT, shown in Figure 3.33, splits merged states whose constituents do not agree on a single successor state for a particular character. When SPLIT terminates, we know that the states that remain merged are equivalent in that they always agree on common successors.

Returning to the example, we initially have states $\{1, 2, 3, 5, 6\}$ and $\{4, 7\}$. Invoking SPLIT, we first observe that states 3 and 6 have a common successor under c and states 1, 2, and 5 have no successor under c (or, equivalently, they have the error state s_E). This forces a split that yields $\{1, 2, 5\}$, $\{3, 6\}$, and $\{4, 7\}$. Now, for character b states 2 and 5 go to the merged state $\{3, 6\}$, but state 1 does not, so another split occurs. We now have $\{1\}$, $\{2, 5\}$, $\{3, 6\}$, and $\{4, 7\}$. At this point, all constituents of merged states agree on the same successor for each input symbol, so we are done.

Once SPLIT is executed, we are essentially done. Transitions between merged states are the same as the transitions between states in the original DFA. That is, if there was a transition between states s_i and s_j under character c , there is now a transition under c from the merged state containing s_i to the merged state containing s_j . The start state is that merged state that contains the original start state. An accepting state is a merged state that contains accepting states (recall that accepting and nonaccepting states are never merged).

Returning to the example, the minimum state automaton we obtain is shown in Figure 3.34.

A proof of the correctness and optimality of this minimization algorithm can be found in most texts on automata theory, such as [HU79].

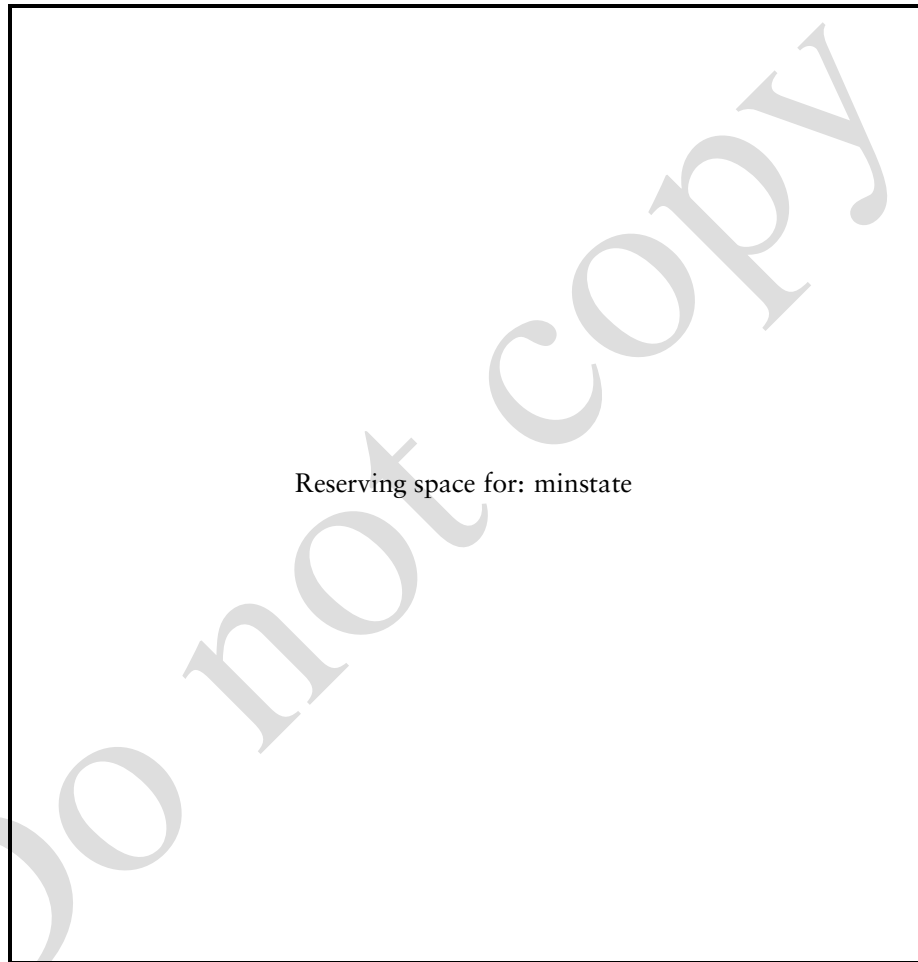


Figure 3.34: The minimum state automaton obtained.

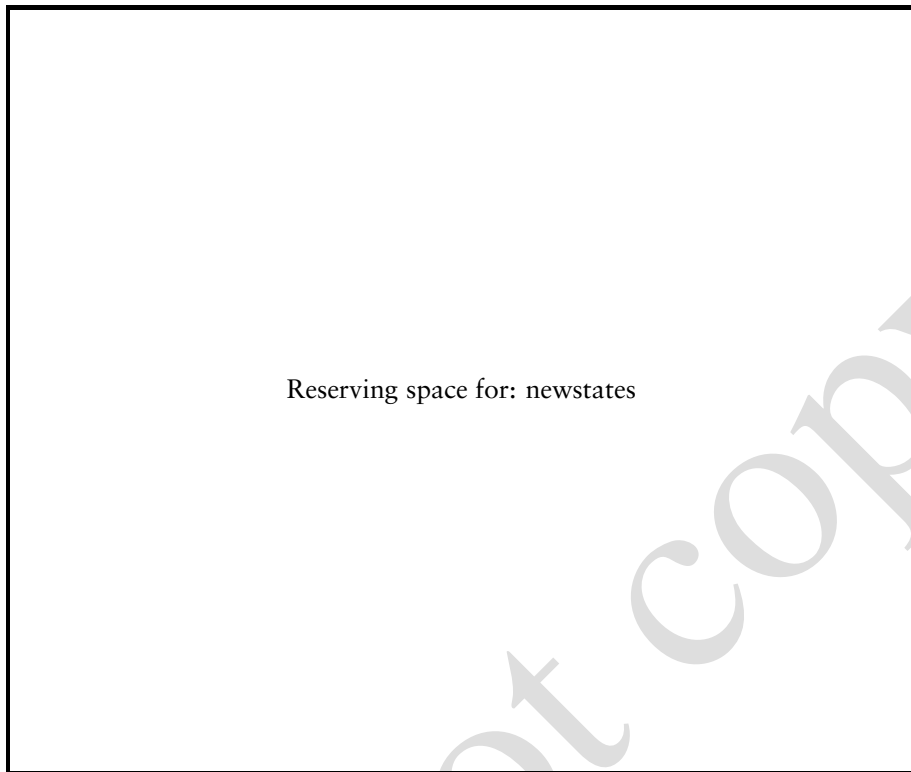


Figure 3.35: An FA with new start and accepting states added.

Translating Finite Automata into Regular Expressions

So far, we have concentrated on the process of converting a given regular expression into an equivalent FA. This is the key step in Lex's construction of a scanner from a set of regular expression token patterns.

Since regular expressions, DFAs, and NFAs are interconvertible, it is also possible to derive for any FA a regular expression that describes the strings that the FA matches. In this section, we briefly discuss an algorithm that does this derivation. This algorithm is sometimes useful when you already have an FA you want to use but you need a regular expression to program Lex or to describe the FA's effect. This algorithm also helps you to see that regular expressions and FAs really are equivalent.

The algorithm we use is simple and elegant. We start with an FA and simplify it by removing states, one-by-one. Simplified FAs are equivalent to the original, except that transitions are now labeled with regular expressions rather than individual characters. We continue removing states until we have an FA with a single transition from the start state to a single accepting state. The regular expression that labels that single transition correctly describes the effect of the original FA.

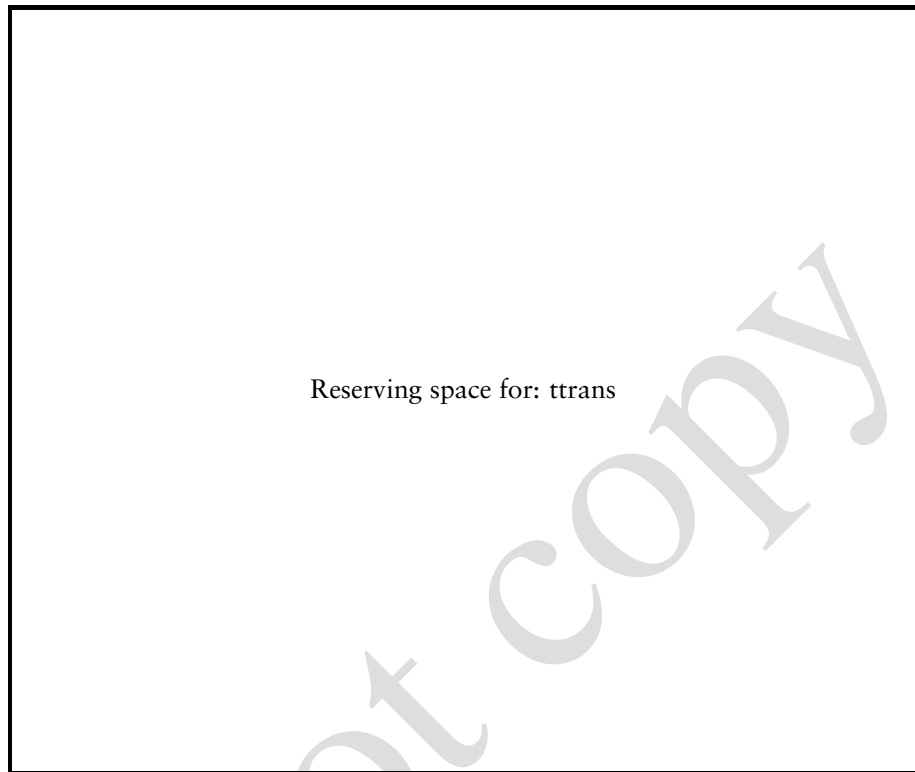


Figure 3.36: The $T1$, $T2$, and $T3$ transformations.

To start, we assume our FA has a start state with no transitions into it and a single accepting state with no transitions out of it. If it doesn't meet these requirements, we can easily transform it by adding a new start state and a new accepting state linked to the original automaton with λ transitions. This is illustrated in Figure 3.35 using the FA we created with `MAKEDETERMINISTIC` in Section 3.7.2. We define three simple transformations, $T1$, $T2$, and $T3$, that will allow us to progressively simplify FAs. The first, illustrated in Figure 3.36(a), notes that if there are two different transitions between the same pair of states, with one transition labeled R and the other labeled S , then we can replace the two transitions with a new one labeled $R | S$. $T1$ simply reflects that we can choose to use either the first transition *or* the second.

Transformation $T2$, illustrated in Figure 3.36(b) allows us to *by pass* a state. That is, if state s has a transition to state r labeled X and state r has a transition to state u labeled Y , then we can go directly from state s to state u with a transition labeled XY .

Transformation $T3$, illustrated in Figure 3.36(c), is similar to transformation $T2$. It, too, allows us to bypass a state. Suppose state s has a transition to state r labeled X and state r has a transition to itself labeled Z , as well as a transition

to state u labeled Y . We can go directly from state s to state u with a transition labeled XZ^*Y . The Z^* term reflects that once we reach state r , we can cycle back into r zero or more times before finally proceeding to u .

We use transformations $T2$ and $T3$ as follows. We consider, in turn, each pair of predecessors and successors a state s has and use $T2$ or $T3$ to link a predecessor state directly to a successor state. In this case, s is no longer needed—all paths through the FA can bypass it! Since s isn't needed, we remove it. The FA is now simpler because it has one fewer states. By removing all states other than the start state and the accepting state (using transformation $T1$ when necessary), we will reach our goal. We will have an FA with only one transition, and the label on this transition will be the regular expression we want. FINDRE, shown in Figure 3.37, implements this algorithm. The algorithm begins by invoking AUGMENT, which introduces new start and accept states. The loop at Step 1 considers each state s of the FA in turn. Transformation $T1$ ensures that each pair of states is connected by at most one transition. This transformation is performed prior to processing s at Step 3. State s is then eliminated by considering the cross-product of states with edges to and from s . For each such pair of states, transformation $T2$ or $T3$ is applied. State s is then removed at Step 2, along with all edges to or from state s . When the algorithm terminates, the only states left are *NewStart* and *NewAccept*, introduced by AUGMENT. The regular expression for the FA labels the transition between these two states.

As an example, we find the regular expression corresponding to the FA in Section 3.7.2. The original FA, with a new start state and accepting state added, is shown in Figure 3.38(a). State 1 has a single predecessor, state 0, and a single successor, state 2. Using a $T3$ transformation, we add an arc directly from state 0 to state 2 and remove state 1. This is shown in Figure 3.38(b). State 2 has a single predecessor, state 0, and three successors, states 2, 4, and 5. Using three $T2$ transformations, we add arcs directly from state 0 to states 3, 4, and 5. State 2 is removed. This is shown in Figure 3.38(c).

State 4 has two predecessors, states 0 and 3. It has one successor, state 5. Using two $T2$ transformations, we add arcs directly from states 0 and 3 to state 5. State 4 is removed. This is shown in Figure 3.38(d). Two pairs of transitions are merged using $T1$ transformations to produce the FA in Figure 3.38(e). Finally, state 3 is bypassed with a $T2$ transformation and a pair of transitions are merged with a $T1$ transformation, as shown in Figure 3.38(f). The regular expression we obtain is

$$b^*ab(a \mid b \mid \lambda) \mid b^*aa \mid b^*a.$$

By expanding the parenthesized subterm and then factoring a common term, we obtain

$$b^*aba \mid b^*abb \mid b^*ab \mid b^*aa \mid b^*a \equiv b^*a(ba \mid bb \mid b \mid a \mid \lambda).$$

Careful examination of the original FA verifies that this expression correctly describes it.

```

function FINDRE(N) : RegExpr
  OrigStates ← N.States
  call AUGMENT(N)
  foreach s ∈ OrigStates do 1
    call ELIMINATE(s)
    N.States ← N.States − { s } 2
  /* return the regular expression labeling the only remaining transition */
end
procedure ELIMINATE(s)
  foreach (x, y) ∈ N.States × N.States | COUNTTRANS(x, y) > 1 do 3
    /* Apply transformation T1 to x and y */
  foreach p ∈ PREDS(s) | p ≠ s do
    foreach u ∈ SUCCS(s) | u ≠ s do
      if CountTrans(s, s) = 0
      then /* Apply Transformation T2 to p, s, and u */
      else /* Apply Transformation T3 to p, s, and u */
    end
  end
function COUNTTRANS(x, y) : Integer
  return (number of transitions from x to y)
end
function PREDS(s) : Set
  return ({ p | (∃ a)(N.T(p, a) = s) })
end
function SUCCS(s) : Set
  return ({ u | (∃ a)(N.T(s, a) = u) })
end
procedure AUGMENT(N)
  OldStart ← N.StartState
  NewStart ← NEWSTATE( )
  /* Define N.T(NewStart, λ) = { OldStart } */
  N.StartState ← NewStart
  OldAccepts ← N.AcceptStates
  NewAccept ← NEWSTATE( )
  foreach s ∈ OldAccepts do
    /* Define N.T(s, λ) = { NewAccept } */
  N.AcceptStates ← { NewAccept }
end

```

Figure 3.37: An algorithm to generate a regular expression from an FA.

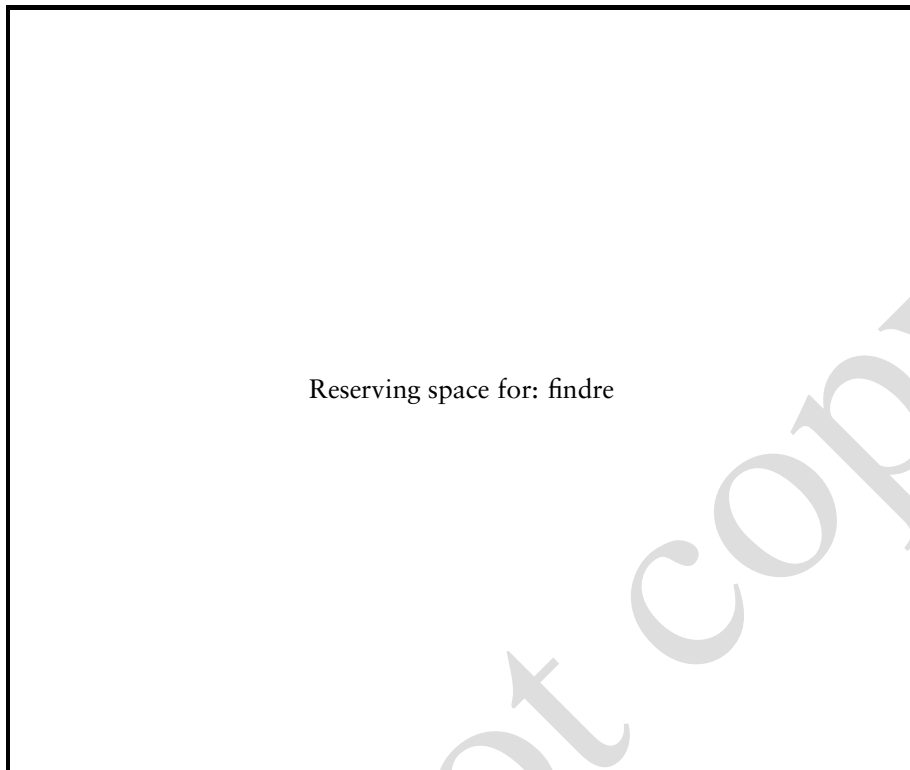


Figure 3.38: Finding a regular expression using FINDRE.

3.8 Summary

We have discussed three equivalent and interchangeable mechanisms for defining tokens: the regular expression, the deterministic finite automaton, and the nondeterministic finite automaton. Regular expressions are convenient for programmers because they allow the specification of token structure without regard for implementation considerations. Deterministic finite automata are useful in implementing scanners because they define token recognition simply and cleanly, on a character-by-character basis. Nondeterministic finite automata form a middle ground. Sometimes they are used for definitional purposes, when it is convenient to simply draw a simple automaton as a “flow diagram” of characters that are to be matched. Sometimes they are directly executed (see Exercise 17), when translation to deterministic finite automata is too costly or inconvenient. Familiarity with all three mechanisms will allow you to use the one best-suited to your needs.

Exercises

1. Assume the following text is presented to a C scanner:

```
main(){
    const float payment = 384.00;
    float bal;
    int month = 0;
    bal=15000;
    while (bal>0){
        printf("Month: %2d Balance: %10.2f\n", month, bal);
        bal=bal-payment+0.015*bal;
        month=month+1;
    }
}
```

What token sequence is produced? For which tokens must extra information be returned in addition to the token code?

2. How many lexical errors (if any) appear in the following C program? How should each error be handled by the scanner?

```
main(){
    if(1<2.)a=1.0else a=1.0e-n;
    subr('aa',"aaaaaa
        aaaaaa");
    /* That's all
}
}
```

3. Write regular expressions that define the strings recognized by the FAs in Figure 3.39.
4. Write DFAs that recognize the tokens defined by the following regular expressions.
- $(a \mid (bc)^*d)^+$
 - $((0 \mid 1)^*(2 \mid 3)^+ \mid 0011)$
 - $(a \text{ Not}(a))^*aaa$
5. Write a regular expression that defines a C-like, fixed-decimal literal with no superfluous leading or trailing zeros. That is, 0.0, 123.01, and 123005.0 are legal, but 00.0, 001.000, and 002345.1000 are illegal.

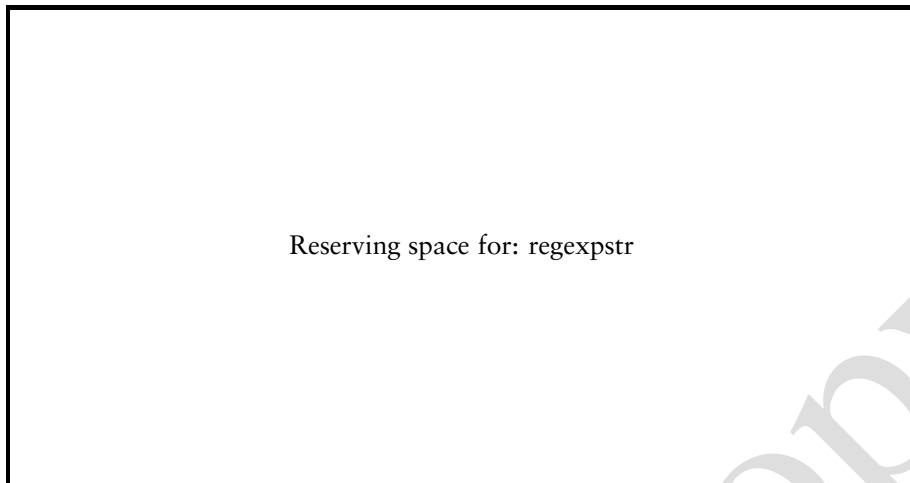


Figure 3.39: FA for Exercise 3.

6. Write a regular expression that defines a C-like comment delimited by `/*` and `*/`. Individual `*`'s and `/`'s may appear in the comment body, but the pair `*/` may not.
7. Define a token class *AlmostReserved* to be those identifiers that are not reserved words but that would be if a single character were changed. Why is it useful to know that an identifier is “almost” a reserved word? How would you generalize a scanner to recognize *AlmostReserved* tokens as well as ordinary reserved words and identifiers?
8. When a compiler is first designed and implemented, it is wise to concentrate on correctness and simplicity of design. After the compiler is fully implemented and tested, you may need to increase compilation speed. How would you determine whether the scanner component of a compiler is a significant performance bottleneck? If it is, what might you do to improve performance (without affecting compiler correctness)?
9. Most compilers can produce a source listing of the program being compiled. This listing is usually just a copy of the source file, perhaps embellished with line numbers and page breaks. Assume you are to produce a prettyprinted listing.
 - (a) How would you modify a Lex scanner specification to produce a prettyprinted listing?
 - (b) How are compiler diagnostics and line numbering complicated when a prettyprinted listing is produced?
10. For most modern programming languages, scanners require little context information. That is, a token can be recognized by examining its text and

perhaps one or two lookahead characters. In Ada, however, additional context is required to distinguish between a single tic (comprising an attribute operator, as in `data'size`) and a tic-character-tic sequence (comprising a quoted character, as in `'x'`). Assume that a Boolean flag `can_parse_char` is set by the parser when a quoted character can be parsed. If the next input character is a tic, `can_parse_char` can be used to control how the tic is scanned. Explain how the `can_parse_char` flag can be cleanly integrated into a Lex-created scanner. The changes you suggest should not unnecessarily complicate or slow the scanning of ordinary tokens.

11. Unlike C, C++, and Java, FORTRAN generally ignores blanks and therefore may need extensive lookahead to determine how to scan an input line. We noted earlier a famous example of this: `DO 10 I = 1 , 10`, which produces seven tokens, in contrast with `DO 10 I = 1 . 10`, which produces three tokens.
 - (a) How would you design a scanner to handle the extended lookahead that FORTRAN requires?
 - (b) Lex contains a mechanism for doing lookahead of this sort. How would you match the identifier (`DO10I`) in this example?
12. Because FORTRAN generally ignores blanks, a character sequence containing n blanks can be scanned as many as 2^n different ways. Are each of these alternatives equally probable? If not, how would you alter the design you proposed in Exercise 11 to examine the most probable alternatives first?
13. You are to design the ultimate programming language, "Utopia 2010." You have already specified the language's tokens using regular expressions and the language's syntax using a CFG. Now you want to determine those token pairs that require whitespace to separate them (such as `else a`) and those that require extra lookahead during scanning (such as `10.0e-22`). Explain how you could use the regular expressions and CFG to automatically find all token pairs that need special handling.
14. Show that the set $\{[{}^k]{}^k \mid k \geq 1\}$ is not regular. *Hint*: Show that no fixed number of FA states is sufficient to exactly match left and right brackets.
15. Show the NFA that would be created for the following expression using the techniques of Section 3.7:
 $(ab^*c) \mid (abc^*)$
 Using MAKEDETERMINISTIC, translate the NFA into a DFA. Using the techniques of Section 3.7.3, optimize the DFA you created into a minimal state equivalent.
16. Consider the following regular expression:
 $(0 \mid 1)^*0(0 \mid 1)(0 \mid 1)(0 \mid 1) \dots (0 \mid 1)$

Display the NFA corresponding to this expression. Show that the equivalent DFA is *exponentially* bigger than the NFA you presented.

17. Translation of a regular expression into an NFA is fast and simple. Creation of an equivalent DFA is slower and can lead to a much larger automaton. An interesting alternative is to scan using NFAs, thus obviating the need to ever build a DFA. The idea is to mimic the operation of the `CLOSE` and `MAKEDETERMINISTIC` routines (as defined in Section 3.7.2) while scanning. A set of possible states, rather than a single current state, is maintained. As characters are read, transitions from each state in the current set are followed, thereby creating a new set of states. If any state in the current set is final, the characters read will comprise a valid token.
Define a suitable encoding for an NFA (perhaps a generalization of the transition table used for DFAs) and write a scanner driver that can use this encoding by following the set-of-states approach outlined previously. This approach to scanning will surely be slower than the standard approach, which uses DFAs. Under what circumstances is scanning using NFAs attractive?
18. Assume e is any regular expression. \bar{e} represents the set of all strings not in the regular set defined by e . Show that \bar{e} is a regular set.
Hint: If e is a regular expression, there is an FA that recognizes the set defined by e . Transform this FA into one that will recognize \bar{e} .
19. Let Rev be the operator that reverses the sequence of characters within a string. For example, $Rev(abc) = cba$. Let R be any regular expression. $Rev(R)$ is the set of strings denoted by R , with each string reversed. Is $Rev(R)$ a regular set? Why?
20. Prove that the DFA constructed by `MAKEDETERMINISTIC` in Section 3.7.2 is equivalent to the original NFA. To do so, you must show that an input string can lead to a final state in the NFA if, and only if, that same string will lead to a final state in the corresponding DFA.
21. You have scanned an integer literal into a character buffer (perhaps `yytext`). You now want to convert the string representation of the literal into numeric (`int`) form. However, the string may represent a value too large to be represented in `int` form. Explain how to convert a string representation of an integer literal into numeric form with full overflow checking.
22. Write lex regular expressions (using character classes if you wish) that match the following sets of strings:
 - (a) The set of all unprintable ASCII characters (those before the blank and the very last character)
 - (b) The string `[""]` (that is, a left bracket, three double quotes, and a right bracket)

- (c) The string $x^{12,345}$ (your solution should be far less than 12,345 characters in length)
23. Write a Lex program that examines the words in an ASCII file and lists the ten most frequently used words. Your program should ignore case and should ignore words that appear in a predefined “don't care” list.
- What changes in your program are needed to make it recognize singular and plural nouns (for example, cat and cats) as the same word? How about different verb tenses (walk versus walked versus walking)?
24. Let *Double* be the set of strings defined as $\{s \mid s = ww\}$. *Double* contains only strings composed of two identical repeated pieces. For example, if you have a vocabulary of the ten digits 0 to 9, then the following strings (and many more!) are in *Double*: 11, 1212, 123123, 767767, 98769876,
- Assume you have a vocabulary consisting only of the single letter *a*. Is *Double* a regular set? Why?
- Assume you now have a vocabulary consisting of the two letters, *a* and *b*. Is *Double* a regular set? Why?
25. Let $Seq(x, y)$ be the set of all strings (of length 1 or more) composed of alternating *x*'s and *y*'s. For example, $Seq(a, b)$ contains *a*, *b*, *ab*, *ba*, *aba*, *bab*, *abab*, *baba*, and so on.
- Write a regular expression that defines $Seq(x, y)$.
- Let *S* be the set of all strings (of length 1 or more) composed of *a*'s, *b*'s, and *c*'s that start with an *a* and in which no two adjacent characters are equal. For example, *S* contains *a*, *ab*, *abc*, *abca*, *acab*, *acac*, ... but not *c*, *aa*, *abb*, *abcc*, *aab*, *cac*, ... Write a regular expression that defines *S*. You may use $Seq(x, y)$ within your regular expression if you wish.
26. Let *AllButLast* be a function that returns all of a string but its last character. For example, $AllButLast(abc) = ab$. $AllButLast(\lambda)$ is undefined. Let *R* be any regular expression that does not generate λ . $AllButLast(R)$ is the set of strings denoted by *R*, with *AllButLast* applied to each string. Thus $AllButLast(a^+b) = a^+$. Show that $AllButLast(R)$ is a regular set.
27. Let *F* be any NFA that contains λ transitions. Write an algorithm that transforms *F* into an equivalent NFA *F'* that contains no λ transitions.
- Note:* You need not use the subset construction, since you're creating a NFA, not a DFA.
28. Let *s* be a string. Define $Insert(s)$ to be the function that inserts a # into each possible position in *s*. If *s* is *n* characters long, $Insert(s)$ returns a set of *n* + 1 strings (since there are *n* + 1 places, a # may be inserted in a string of length *n*).

For example, $Insert(abc) = \{ \#abc, a\#bc, ab\#c, abc\# \}$. *Insert* applied to a set of strings is the union of *Insert* applied to members of the set. Thus $Insert(ab, de) = \{ \#ab, a\#b, ab\#, \#de, d\#e, de\# \}$.

Let R be any regular set. Show that $Insert(R)$ is a regular set.

Hint: Given an FA for R , construct one for $Insert(R)$.

29. Let D be any deterministic finite automaton. Assume that you know D contains exactly n states and that it accepts at least one string of length n or greater. Show that D must also accept at least one string of length $2n$ or greater.

Do not copy

Do not copy

Bibliography

- [AMT89] Andrew W. Appel, James S. Mattson, and David R. Tarditi. *A lexical analyzer generator for Standard ML*. Princeton University, 1989. distributed with SML/NJ software.
- [BC93] Peter Bumbulis and Donald D. Cowan. Re2c: a more versatile scanner generator. *ACM Letters on Programming Languages and Systems*, 2(1-4):70–84, 1993.
- [Ber97] Elliot Berk. Jlex: A lexical analyzer generator for java. Princeton University, available at <http://www.cs.princeton.edu/appel/modern/java/JLex/manual.html>, 1997.
- [Cic86] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23:17–19, 1986.
- [Gra88] Robert W. Gray. γ -gla: A generator for lexical analyzers that programmers can use. *Summer Usenix Conference Proceedings*, 1988.
- [Gro89] J. Grosch. Efficient generation of lexical analysers. *Software – Practice and Experience*, 19:1089–1103, 1989.
- [HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [Jac87] Van Jacobsen. Tuning unix lex, or it's not true what they say about lex. *Winter Usenix Conference Proceedings*, pages 163–164, 1987.
- [LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex and yacc*. O'Reilly and Associates, 1992.
- [LS75] M.E. Lesk and E. Schmidt. Lex—a lexical analyzer generator. *Unix Programmer's Manual 2*, 1975.
- [Moe91] H. Moessenboeck. A generator for production quality compilers. *Proc. of the 3rd int. workshop on compiler compilers*, 1991. Lecture Notes in Computer Science 477.
- [NF88] T. Nguyen and K. Forester. Alex – an ada lexical analysis generator. Arcadia Document UCI-88-17, University of California, Irvine, 1988.

- [Pax88] V. Paxson. Flex man pages. In ftp distribution from `//ftp.ee.lbl.gov`, 1988.
- [PDC89] T.J. Parr, H.G. Dietz, and W.E. Cohen. *PCCTS Reference Manual*. Purdue University, 1989.

Do not copy

4

Grammars and Parsing

Formally a **language** is a set of finite-length strings over a finite alphabet. Because most interesting languages are infinite sets, we cannot define such languages by enumerating their elements. A **grammar** is a compact, finite representation of a language. In Chapter Chapter:global:two, we discuss the rudiments of **context-free grammars** (CFGs) and define a simple language using a CFG. Due to their efficiency and precision, CFGs are commonly used for defining the syntax of actual programming languages. In Chapter 4, we formalize the definition and notation for CFGs and present algorithms that analyze such grammars in preparation for the parsing techniques covered in Chapters Chapter:global:five and Chapter:global:six.

4.1 Context-Free Grammars: Concepts and Notation

A CFG is defined by the following four components.

1. A finite **terminal alphabet** Σ . This is the set of tokens produced by the scanner. We always augment this set with the token $\$$, which signifies end-of-input.
2. A finite **nonterminal alphabet** N . Symbols in this alphabet are *variables* of the grammar.
3. A **start symbol** $S \in N$ that initiates all *derivations*. S is also called the **goal symbol**.
4. P , a finite set of productions (sometimes called **rewriting rules**) of the form $A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_m$, where $A \in N$, $\mathcal{X}_i \in N \cup \Sigma$, $1 \leq i \leq m$, and $m \geq 0$. The

only valid production with $m = 0$ is of the form $A \rightarrow \lambda$, where λ denotes the **empty string**.

These components are often grouped into a four-tuple, $G = (N, \Sigma, P, S)$, which is the formal definition of a CFG. The terminal and nonterminal alphabets must be disjoint (*i.e.*, $\Sigma \cap N = \emptyset$). The **vocabulary** V of a CFG is the set of terminal and nonterminal symbols (*i.e.*, $V = \Sigma \cup N$).

A CFG is essentially a recipe for creating strings. Starting with S , nonterminals are rewritten using the grammar's productions until only terminals remain. A rewrite using the production $A \rightarrow \alpha$ replaces the nonterminal A with the vocabulary symbols in α . As a special case, a rewrite using the production $A \rightarrow \lambda$ causes A to be erased. Each rewrite is a step in a **derivation** of the resulting string. The set of terminal strings derivable from S comprises the **context-free language** of grammar G , denoted $L(G)$.

In describing parsers, algorithms, and grammars, consistency is useful in denoting symbols and strings of symbols. We follow the notation set out in the following chart.

Names Beginning With	Represent Symbols In	Examples
Uppercase	N	A, B, C, Prefix
Lowercase and punctuation	Σ	a, b, c, if, then, (, ;
\mathcal{X}, \mathcal{Y}	$N \cup \Sigma$	$\mathcal{X}_i, \mathcal{Y}_3$
Other Greek letters	$(N \cup \Sigma)^*$	α, γ

Using this notation, we write a production as $A \rightarrow \alpha$ or $A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_m$, depending on whether the detail of the production's **right-hand side** (RHS) is of interest. This format emphasizes that a production's **left-hand side** (LHS) must be a single nonterminal but the RHS is a string of zero or more vocabulary symbols.

There is often more than one way to rewrite a given nonterminal; in such cases, multiple productions share the same LHS symbol. Instead of repeating the LHS symbol, an “or notation” is used.

$$\begin{array}{l}
 A \rightarrow \alpha \\
 \quad | \quad \beta \\
 \quad \dots \\
 \quad | \quad \zeta
 \end{array}$$

This is an abbreviation for the following sequence of productions.

$$\begin{array}{l}
 A \rightarrow \alpha \\
 A \rightarrow \beta \\
 \dots \\
 A \rightarrow \zeta
 \end{array}$$

If $A \rightarrow \gamma$ is a production, then $\alpha A \beta \Rightarrow \alpha \gamma \beta$ denotes one step of a derivation using this production. We extend \Rightarrow to \Rightarrow^+ (derived in one or more steps) and \Rightarrow^* (derived in zero or more steps). If $S \Rightarrow^* \beta$, then β is said to be a *sentential*

1	E	→	Prefix (E)
			v Tail
3	Prefix	→	f
4			λ
5	Tail	→	+ E
6			λ

Figure 4.1: A simple expression grammar.

form of the CFG. $SF(G)$ denotes the set of sentential forms of grammar G . Thus $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^+ w\}$. Also, $L(G) = SF(G) \cap \Sigma^*$. That is, the language of G is simply those sentential forms of G that are terminal strings.

Throughout a derivation, if more than one nonterminal is present in a sentential form, then there is a choice as to which nonterminal should be expanded in the next step. Thus to characterize a derivation sequence, we need to specify, at each step, which nonterminal is expanded and which production is applied. We can simplify this characterization by adopting a convention such that nonterminals are rewritten in some systematic order. Two such conventions are the

- leftmost derivation and
- rightmost derivation.

4.1.1 Leftmost Derivations

A derivation that always chooses the *leftmost* possible nonterminal at each step is called a **leftmost derivation**. If we know that a derivation is leftmost, we need only specify the productions in order of their application; the expanded nonterminal is implicit. To denote derivations that are leftmost, we use \Rightarrow_{lm} , \Rightarrow_{lm}^+ , and \Rightarrow_{lm}^* . A sentential form produced via a leftmost derivation is called a **left sentential form**. The production sequence discovered by a large class of parsers—the top-down parsers—is a leftmost derivation. Hence, these parsers are said to produce a *leftmost parse*.

As an example, consider the grammar shown in Figure 4.1, which generates simple expressions (v represents a variable and f represents a function). A leftmost derivation of $f(v + v)$ is as follows.

E	\Rightarrow_{lm}	Prefix (E)
	\Rightarrow_{lm}	f (E)
	\Rightarrow_{lm}	f (v Tail)
	\Rightarrow_{lm}	f (v + E)
	\Rightarrow_{lm}	f (v + v Tail)
	\Rightarrow_{lm}	f (v + v)

4.1.2 Rightmost Derivations

An alternative to a leftmost derivation is a **rightmost derivation** (sometimes called a **canonical derivation**). In such derivations the rightmost possible nonterminal is always expanded. This derivation sequence may seem less intuitive given the English convention of processing information left-to-right, but such derivations are produced by an important class of parsers, namely the bottom-up parsers discussed in Chapter Chapter:global:six.

As a bottom-up parser discovers the productions that derive a given token sequence, it traces a rightmost derivation, but the productions are applied in *reverse order*. That is, the last step taken in a rightmost derivation is the first production applied by the bottom-up parser; the first step involving the start symbol is the parser's final production. The sequence of productions applied by a bottom-up parser is called a **rightmost** or **canonical** parse. For derivations that are rightmost, the notation \Rightarrow_{rm} , \Rightarrow_{rm}^+ , and \Rightarrow_{rm}^* is used. A sentential form produced via a rightmost derivation is called a **right sentential form**. A rightmost derivation of the grammar shown in Figure 4.1 is as follows.

$$\begin{aligned} E &\Rightarrow_{rm} \text{Prefix (E)} \\ &\Rightarrow_{rm} \text{Prefix (v Tail)} \\ &\Rightarrow_{rm} \text{Prefix (v + E)} \\ &\Rightarrow_{rm} \text{Prefix (v + v Tail)} \\ &\Rightarrow_{rm} \text{Prefix (v + v)} \\ &\Rightarrow_{rm} f (v + v) \end{aligned}$$

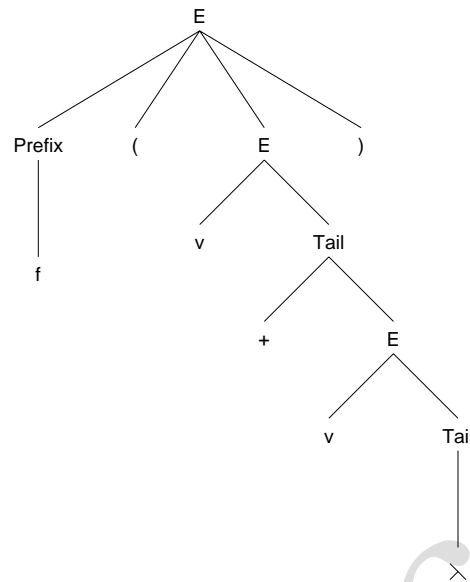
4.1.3 Parse Trees

A derivation is often represented by a *parse tree* (sometimes called a *derivation tree*). A **parse tree** has the following characteristics.

- It is rooted by the grammar's start symbol S .
- Each node is either a grammar symbol or λ .
- Its interior nodes are nonterminals. An interior node and its children represent the application of a production. That is, a node representing a nonterminal A can have offspring $\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_m$ if, and only if, there exists a grammar production $A \rightarrow \mathcal{X}_1 \mathcal{X}_2 \dots \mathcal{X}_m$. When a derivation is complete, each leaf of the corresponding parse tree is either a terminal symbol or λ .

Figure 4.2 shows the parse tree for $f(v + v)$ using the grammar from Figure 4.1. Parse trees serve nicely to visualize how a string is structured by a grammar. A leftmost or rightmost derivation is essentially a textual representation of a parse tree, but the derivation conveys also the order in which the productions are applied.

A sentential form is derivable from a grammar's start symbol. Hence, a parse tree must exist for every sentential form. Given a sentential form and its parse tree, a **phrase** of the sentential form is a sequence of symbols descended from

Figure 4.2: The parse tree for $f(v + v)$.

a single nonterminal in the parse tree. A **simple** or **prime phrase** is a phrase that contains no smaller phrase. That is, it is a sequence of symbols directly derived from a nonterminal. The **handle** of a sentential form is the leftmost simple phrase. (Simple phrases cannot overlap, so “leftmost” is unambiguous.) Consider the parse tree in Figure 4.2 and the sentential form $f(v \text{ Tail})$. f and $v \text{ Tail}$ are simple phrases and f is the handle. Handles are important because they represent individual derivation steps, which can be recognized by various parsing techniques.

4.1.4 Other Types of Grammars

Although CFGs serve well to characterize syntax, most programming languages contain rules that are not expressible using CFGs. For example, the rule that variables must be declared before they are used cannot be expressed, because a CFG provides no mechanism for transmitting to the body of a program the exact set of variables that has been declared. In practice, syntactic details that cannot be represented in a CFG are considered part of the static semantics and are checked by semantic routines (along with scope and type rules). The non-CFGs that are relevant to programming language translation are the

- regular grammars, which are less powerful than CFGs, and the
- context-sensitive and unrestricted grammars, which are more powerful.

Regular Grammars

A CFG that is limited to productions of the form $A \rightarrow a B$ or $C \rightarrow d$ is a **regular grammar**. Each rule's RHS consists of either a symbol from $\Sigma \cup \{\lambda\}$ followed by a nonterminal symbol or just a symbol from $\Sigma \cup \{\lambda\}$. As the name suggests, a regular grammar defines a regular set (see Exercise 13.) We observed in Chapter Chapter:global:three that the language $\{[]^i \mid i \geq 1\}$ is not regular; this language is generated by the following CFG.

$$\begin{array}{lcl} 1 & S & \rightarrow T \\ 2 & T & \rightarrow [T] \\ 3 & & | \lambda \end{array}$$

This grammar establishes that the languages definable by regular grammars (regular sets) are a *proper* subset of the context-free languages.

Beyond Context-Free Grammars

CFGs can be generalized to create richer definitional mechanisms. A **context-sensitive grammar** requires that nonterminals be rewritten only when they appear in a particular context (for example, $\alpha A \beta \rightarrow \alpha \delta \beta$), provided the rule never causes the sentential form to contract in length. An **unrestricted** or **type-0 grammar** is the most general. It allows arbitrary patterns to be rewritten.

Although context-sensitive and unrestricted grammars are more powerful than CFGs, they also are far less useful.

- Efficient parsers for such grammars do not exist. Without a parser, a grammar definition cannot participate in the automatic construction of compiler components.
- It is difficult to prove properties about such grammars. For example, it would be daunting to prove that a given type-0 grammar generates the C programming language.

Efficient parsers for many classes of CFGs do exist. Hence, CFGs present a nice balance between generality and practicality. Throughout this text, we focus on CFGs. Whenever we mention a grammar (without saying which kind), you should assume that the grammar is context-free.

4.2 Properties of CFGs

CFGs are a definitional mechanism for specifying languages. Just as there are many programs that compute the same result, so there are many grammars that generate the same language. Some of these grammars may have properties that complicate parser construction. The most common of these properties are

- the inclusion of useless nonterminals,

- allowing a derived string to have two or more different parse trees, and
- generating the wrong language.

In this section, we discuss these properties and their implication for language processing.

4.2.1 Reduced Grammars

A grammar is **reduced** if each of its nonterminals and productions participates in the derivation of some string in the grammar's language. Nonterminals that can be safely removed are called **useless**.

$$\begin{array}{l} 1 \quad S \rightarrow A \\ 2 \quad \quad | B \\ 3 \quad A \rightarrow a \\ 4 \quad B \rightarrow B b \\ 5 \quad C \rightarrow c \end{array}$$

The above grammar contains two kinds of nonterminals that cannot participate in any derived string.

- With S as the start symbol, the nonterminal C cannot appear in any phrase.
- Any phrase that mentions B cannot be rewritten to contain only terminals.

Exercises 14 and 15 consider how to detect both forms of useless nonterminals. When B, C, and their associated productions are removed, the following reduced grammar is obtained.

$$\begin{array}{l} 1 \quad S \rightarrow A \\ 2 \quad A \rightarrow a \end{array}$$

Many parser generators verify that a grammar is reduced. An unreduced grammar probably contains errors that result from mistyping of grammar specifications.

4.2.2 Ambiguity

Some grammars allow a derived string to have two or more different parse trees (and thus a nonunique structure). Consider the following grammar, which generates expressions using the infix operator for subtraction.

$$\begin{array}{l} 1 \quad \text{Expr} \rightarrow \text{Expr} - \text{Expr} \\ 2 \quad \quad | \text{id} \end{array}$$

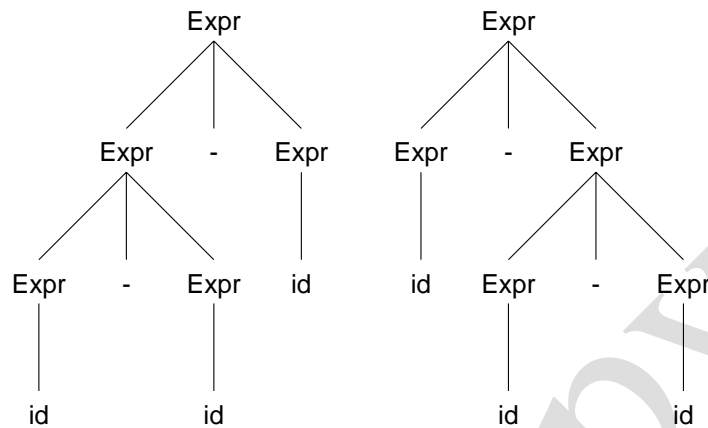


Figure 4.3: Two parse trees for `id - id - id`.

This grammar allows two different parse trees for `id - id - id`, as illustrated in Figure 4.3.

Grammars that allow different parse trees for the same terminal string are called **ambiguous**. They are rarely used because a unique structure (*i.e.*, parse tree) cannot be guaranteed for all inputs. Hence, a unique translation, guided by the parse tree structure, may not be obtained.

It seems we need an algorithm that checks an arbitrary CFG for ambiguity. Unfortunately, no algorithm is possible for this in the general case [HU79, Mar91]. Fortunately, for certain grammar classes, successful parser construction by the algorithms we discuss in Chapters Chapter:global:five and Chapter:global:six proves a grammar to be unambiguous.

4.2.3 Faulty Language Definition

The most potentially serious flaw that a grammar might have is that it generates the “wrong” language. That is, the terminal strings derivable by the grammar do not correspond *exactly* to the strings present in the desired language. This is a subtle point, because a grammar typically serves as the very definition of a language's syntax.

The correctness of a grammar is usually tested informally by attempting to parse a set of inputs, some of which are supposed to be in the language and some of which are not. One might try to compare for equality the languages defined by a pair of grammars (considering one a standard), but this is rarely done. For some grammar classes, such verification is possible; for others, no comparison algorithm is known. A general comparison algorithm applicable to all CFGs is known to be impossible to construct [Mar91].

```

foreach  $p \in Prods$  of the form “ $A \rightarrow \alpha [ \mathcal{X}_1 \dots \mathcal{X}_n ] \beta$ ” do
   $N \leftarrow \text{NEWNONTERM}()$ 
   $p \leftarrow “A \rightarrow \alpha N \beta”$ 
   $Prods \leftarrow Prods \cup \{ “N \rightarrow \mathcal{X}_1 \dots \mathcal{X}_n” \}$ 
   $Prods \leftarrow Prods \cup \{ “N \rightarrow \lambda” \}$ 
foreach  $p \in Prods$  of the form “ $B \rightarrow \gamma \{ \mathcal{X}_1 \dots \mathcal{X}_m \} \delta$ ” do
   $M \leftarrow \text{NEWNONTERM}()$ 
   $p \leftarrow “B \rightarrow \gamma M \delta”$ 
   $Prods \leftarrow Prods \cup \{ “M \rightarrow \mathcal{X}_1 \dots \mathcal{X}_m M” \}$ 
   $Prods \leftarrow Prods \cup \{ “M \rightarrow \lambda” \}$ 

```

Figure 4.4: Algorithm to transform a BNF grammar into standard form.

4.3 Transforming Extended Grammars

Backus-Naur form (BNF) extends the grammar notation defined in Section 4.1 with syntax for defining optional and repeated symbols.

- Optional symbols are enclosed in square brackets. In the production

$$A \rightarrow \alpha [\mathcal{X}_1 \dots \mathcal{X}_n] \beta$$

the symbols $\mathcal{X}_1 \dots \mathcal{X}_n$ are entirely present or absent between the symbols of α and β .

- Repeated symbols are enclosed in braces. In the production

$$B \rightarrow \gamma \{ \mathcal{X}_1 \dots \mathcal{X}_m \} \delta$$

the entire sequence of symbols $\mathcal{X}_1 \dots \mathcal{X}_m$ can be repeated zero or more times.

These extensions are useful in representing many programming language constructs. In Java, declarations can optionally include modifiers such as `final`, `static`, and `const`. Each declaration can include a *list* of identifiers. A production specifying a Java-like declaration could be as follows.

$$\text{Declaration} \rightarrow [\text{final}] [\text{static}] [\text{const}] \text{Type identifier } \{ , \text{identifier} \}$$

This declaration insists that the modifiers be ordered as shown. Exercises 11 and 12 consider how to specify the optional modifiers in any order.

Although BNF can be useful, algorithms for analyzing grammars and building parsers assume the standard grammar notation as introduced in Section 4.1. The algorithm in Figure 4.4 transforms extended BNF grammars into standard form. For the BNF syntax involving braces, the transformation uses *right* recursion on M to allow zero or more occurrences of the symbols enclosed within braces. This transformation also works using left recursion—the resulting grammar would have generated the same language.

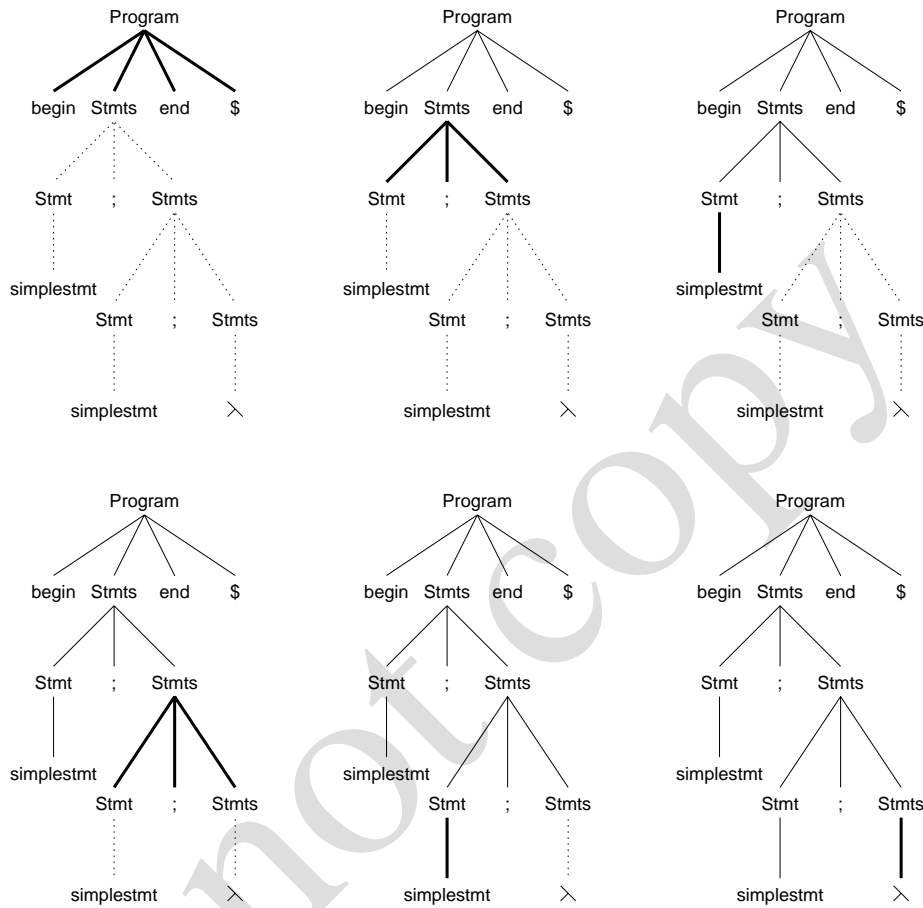


Figure 4.5: A top-down parse of begin simplestmt ; simplestmt ; end \$.

As discussed in Section 4.1, a particular derivation (*e.g.*, leftmost or rightmost) depends on the structure of the grammar. It turns out that right-recursive rules are more appropriate for top-down parsers, which produce leftmost derivations. Similarly, left-recursive rules are more suitable for bottom-up parsers, which produce rightmost derivations.

4.4 Parsers and Recognizers

Compilers are expected to verify the syntactic validity of their inputs with respect to a grammar that defines the programming language's syntax. Given a grammar G and an input string x , the compiler must determine if $x \in L(G)$. An algorithm that performs this test is called a **recognizer**.

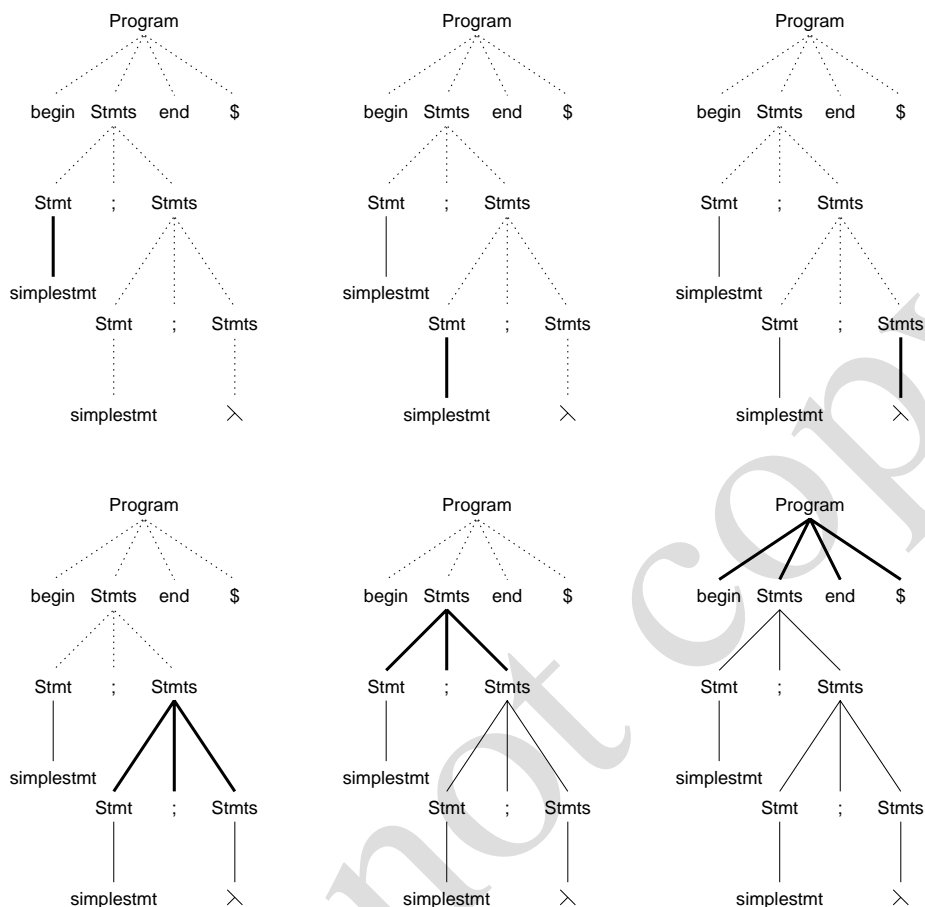


Figure 4.6: A bottom-up parse of begin simplestmt ; simplestmt ; end \$.

For language translation, we must determine not only the string's validity, but also its structure, or **parse tree**. An algorithm for this task is called a **parser**. Generally, there are two approaches to parsing.

- A parser is considered **top-down** if it generates a parse tree by starting at the root of the tree (the start symbol), expanding the tree by applying productions in a depth-first manner. A top-down parse corresponds to a preorder traversal of the parse tree. Top-down parsing techniques are *predictive* in nature because they always predict the production that is to be matched before matching actually begins. The top-down approach includes the recursive-descent parser discussed in Chapter Chapter:global:two.
- The **bottom-up** parsers generate a parse tree by starting at the tree's leaves and working toward its root. A node is inserted in the tree only after its

children have been inserted. A bottom-up parse corresponds to a postorder traversal of the parse tree.

The following grammar generates the skeletal block structure of a programming language.

1	Program	→	begin Stmts end \$
2	Stmts	→	Stmt ; Stmts
3			λ
4	Stmt	→	simplestmt
5			begin Stmts end

Using this grammar, Figures 4.5 and 4.6 illustrate a top-down and bottom-up parse of the string `begin simplestmt ; simplestmt ; end $`.

When specifying a parsing technique, we must state whether a leftmost or rightmost parse will be produced. The best-known and most widely used top-down and bottom-up parsing strategies are called LL and LR, respectively. These names seem rather arcane, but they reflect how the input is processed and which kind of parse is produced. In both cases, the first character (L) states that the token sequence is processed from left to right. The second letter (L or R) indicates whether a leftmost or rightmost parse is produced. The parsing technique can be further characterized by the number of lookahead symbols (*i.e.*, symbols beyond the current token) that the parser may consult to make parsing choices. LL(1) and LR(1) parsers are the most common, requiring only one symbol of lookahead.

4.5 Grammar Analysis Algorithms

It is often necessary to analyze a grammar to determine if it is suitable for parsing and, if so, to construct tables that can drive a parsing algorithm. In this section, we discuss a number of important analysis algorithms, and so strengthen the basic concepts of grammars and derivations. These algorithms are central to the automatic construction of parsers, as discussed in Chapters Chapter:global:five and Chapter:global:six.

4.5.1 Grammar Representation

The algorithms presented in this chapter refer to a collection of utilities for accessing and modifying representations of a CFG. The efficiency of these algorithms is affected by the data structures upon which these utilities are built. In this section, we examine how to represent CFGs efficiently. We assume that the implementation programming language offers the following constructs directly or by augmentation.

- A **set** is an unordered collection of distinct objects.
- A **list** is an ordered collection of objects. An object can appear multiple times in a list.

- An **iterator** is a construct that enumerates the contents of a set or list.

As discussed in Section 4.1, a grammar formally contains two disjoint sets of symbols, Σ and N , which contain the grammar's terminals and nonterminals, respectively. Grammars also contain a designated start symbol and a set of productions. The following observations are relevant to obtaining an efficient representation for grammars.

- Symbols are rarely deleted from a grammar.
- Transformations such as those shown in Figure 4.4 can add symbols and productions to a grammar.
- Grammar-based algorithms typically visit all rules for a given nonterminal or visit all occurrences of a given symbol in the productions.
- Most algorithms process a production's RHS one symbol at a time.

Based on these observations, we represent a production by its LHS and a list of the symbols on its RHS. The empty string λ is not represented explicitly as a symbol. Instead, a production $A \rightarrow \lambda$ has an empty list of symbols for its RHS. The collection of grammar utilities is as follows.

GRAMMAR(S): Creates a new grammar with start symbol S and no productions.

PRODUCTION(A, rhs): Creates a new production for nonterminal A and returns a descriptor for the production. The iterator rhs supplies the symbols for the production's RHS.

PRODUCTIONS(): Returns an iterator that visits each production in the grammar.

NONTERMINAL(A): Adds A to the set of nonterminals. An error occurs if A is already a terminal symbol. The function returns a descriptor for the nonterminal.

TERMINAL(x): Adds x to the set of terminals. An error occurs if x is already a nonterminal symbol. The function returns a descriptor for the terminal.

NONTERMINALS(): Returns an iterator for the set of nonterminals.

TERMINALS(): Returns an iterator for the set of terminal symbols.

ISTERMINAL(\mathcal{X}): Returns **true** if \mathcal{X} is a terminal; otherwise, returns **false**.

RHS(p): Returns an iterator for the symbols on the RHS of production p .

LHS(p): Returns the nonterminal defined by production p .

PRODUCTIONSFOR(A): Returns an iterator that visits each production for nonterminal A .

```

procedure DERIVESEMPTYSTRING( )
  foreach  $A \in \text{NONTERMINALS}( )$  do  $\text{SymbolDerivesEmpty}(A) \leftarrow \text{false}$ 
  foreach  $p \in \text{PRODUCTIONS}( )$  do
     $\text{RuleDerivesEmpty}(p) \leftarrow \text{false}$ 
    call COUNTSYMBOLS( $p$ ) 1
    call CHECKFOREMPTY( $p$ )
    foreach  $\mathcal{X} \in \text{WorkList}$  do 2
       $\text{WorkList} \leftarrow \text{WorkList} - \{ \mathcal{X} \}$  3
      foreach  $x \in \text{OCCURRENCES}(\mathcal{X})$  do 4
         $p \leftarrow \text{PRODUCTION}(x)$ 
         $\text{Count}(p) \leftarrow \text{Count}(p) - 1$ 
        call CHECKFOREMPTY( $p$ )
      end
    end
  procedure COUNTSYMBOLS( $p$ )
     $\text{Count}(p) \leftarrow 0$ 
    foreach  $\mathcal{X} \in \text{RHS}(p)$  do  $\text{Count}(p) \leftarrow \text{Count}(p) + 1$ 
  end
  procedure CHECKFOREMPTY( $p$ )
    if  $\text{Count}(p) = 0$ 
    then
       $\text{RuleDerivesEmpty}(p) \leftarrow \text{true}$  5
       $A \leftarrow \text{LHS}(p)$ 
      if not  $\text{SymbolDerivesEmpty}(A)$ 
      then
         $\text{SymbolDerivesEmpty}(A) \leftarrow \text{true}$  6
         $\text{WorkList} \leftarrow \text{WorkList} \cup \{ A \}$  7
    end
  end

```

Figure 4.7: Algorithm for determining nonterminals and productions that can derive λ .

OCCURRENCES(\mathcal{X}): Returns an iterator that visits each occurrence of \mathcal{X} in the RHS of all rules.

PRODUCTION(y): Returns a descriptor for the production $A \rightarrow \alpha$ where α contains the occurrence y of some vocabulary symbol.

TAIL(y): Accesses the symbols appearing after an occurrence. Given a symbol occurrence y in the rule $A \rightarrow \alpha y \beta$, **TAIL**(y) returns an iterator for the symbols in β .

4.5.2 Deriving the Empty String

One of the most common grammar computations is determining which nonterminals can derive λ . This information is important because such nonterminals may disappear during a parse and hence must be carefully handled. Determining if a nonterminal can derive λ is not entirely trivial because the derivation can take more than one step:

$$A \Rightarrow BCD \Rightarrow BC \Rightarrow B \Rightarrow \lambda.$$

An algorithm to compute the productions and symbols that can derive λ is shown in Figure 4.7. The computation utilizes a *worklist* at Step 2. A **worklist** is a set that is augmented and diminished as the algorithm progresses. The algorithm is finished when the worklist is empty. Thus the loop at Step 2 must account for changes to the set *WorkList*. To prove termination of algorithms that utilize worklists, it must be shown that all worklist elements appear a finite number of times.

In the algorithm of Figure 4.7, the worklist contains nonterminals that are discovered to derive λ . The integer $Count(p)$ is initialized at Step 1 to the number of symbols on p 's RHS. The count for any production of the form $A \rightarrow \lambda$ is 0. Once a production is known to derive λ , its LHS is placed on the worklist at Step 7. When a symbol is taken from the worklist at Step 3, each occurrence of the symbol is visited at Step 4 and the count of the associated production is decremented by 1. This process continues until the worklist is exhausted. The algorithm establishes two structures related to derivations of λ , as follows.

- $RuleDerivesEmpty(p)$ indicates whether production p can derive λ . When every symbol in rule p 's RHS can derive λ , Step 5 establishes that p can derive λ .
- $SymbolDerivesEmpty(A)$ indicates whether the nonterminal A can derive λ . When any production for A can derive λ , Step 6 establishes that A can derive λ .

Both forms of information are useful in the grammar analysis and parsing algorithms discussed in Chapters 4, Chapter:global:five, and Chapter:global:six.

4.5.3 First Sets

A set commonly consulted by parser generators is $First(\alpha)$. This is the set of all terminal symbols that can begin a sentential form derivable from the string of grammar symbols in α . Formally,

$$First(\alpha) = \{ a \in \Sigma \mid \alpha \Rightarrow^* a \beta \}.$$

Some texts include λ in $First(\alpha)$ if $\alpha \Rightarrow^* \lambda$. The resulting algorithms require frequent subtraction of λ from symbol sets. We adopt the convention of *never*

```

function FIRST( $\alpha$ ) : Set
  foreach  $A \in \text{NONTERMINALS}()$  do  $\text{VisitedFirst}(A) \leftarrow \text{false}$       8
   $\text{ans} \leftarrow \text{INTERNALFIRST}(\alpha)$ 
  return ( $\text{ans}$ )
end
function INTERNALFIRST( $\mathcal{X}\beta$ ) : Set
  if  $\mathcal{X}\beta = \perp$                                                          9
  then return ( $\emptyset$ )
  if  $\mathcal{X} \in \Sigma$                                                          10
  then return ( $\{\mathcal{X}\}$ )
  /*                                                                    */
  /*           $\mathcal{X}$  is a nonterminal.                                     */ 11
  /*                                                                    */
   $\text{ans} \leftarrow \emptyset$ 
  if not  $\text{VisitedFirst}(\mathcal{X})$ 
  then
     $\text{VisitedFirst}(\mathcal{X}) \leftarrow \text{true}$                                   12
    foreach  $\text{rhs} \in \text{ProductionsFor}(\mathcal{X})$  do
       $\text{ans} \leftarrow \text{ans} \cup \text{INTERNALFIRST}(\text{rhs})$                     13
  if  $\text{SymbolDerivesEmpty}(\mathcal{X})$                                           14
  then  $\text{ans} \leftarrow \text{ans} \cup \text{INTERNALFIRST}(\beta)$ 
  return ( $\text{ans}$ )                                                         15
end

```

Figure 4.8: Algorithm for computing $\text{First}(\alpha)$.

including λ in $\text{First}(\alpha)$. Testing whether a given string of symbols α derives λ is easily accomplished—when the results from the algorithm of Figure 4.7 are available.

$\text{First}(\alpha)$ is computed by scanning α left-to-right. If α begins with a terminal symbol a , then clearly $\text{First}(\alpha) = \{a\}$. If a nonterminal symbol A is encountered, then the grammar productions for A must be consulted. Nonterminals that can derive λ potentially disappear during a derivation, so the computation must account for this as well.

As an example, consider the nonterminals *Tail* and *Prefix* from the grammar in Figure 4.1. Each nonterminal has one production that contributes information directly to the nonterminal's First set. Each nonterminal also has a λ -production, which contributes nothing. The solutions are as follows.

$$\begin{aligned} \text{First}(\text{Tail}) &= \{+\} \\ \text{First}(\text{Prefix}) &= \{f\} \end{aligned}$$

In some situations, the First set of one symbol can depend on the First sets of other symbols. To compute $\text{First}(E)$, the production $E \rightarrow \text{Prefix}(E)$ requires com-

putation of $\text{First}(\text{Prefix})$. Because $\text{Prefix} \Rightarrow^* \lambda$, $\text{First}(\epsilon)$ must also be included. The resulting set is as follows.

$$\text{First}(E) = \{v, f, \{\}$$

Termination of $\text{First}(A)$ must be handled properly in grammars where the computation of $\text{First}(A)$ appears to depend on $\text{First}(A)$, as follows.

$$\begin{array}{l} A \rightarrow B \\ \quad \dots \\ B \rightarrow C \\ \quad \dots \\ C \rightarrow A \end{array}$$

In this grammar, $\text{First}(A)$ depends on $\text{First}(B)$, which depends on $\text{First}(C)$, which depends on $\text{First}(A)$. In computing $\text{First}(A)$, we must avoid endless iteration or recursion. A sophisticated algorithm could preprocess the grammar to determine such cycles of dependence. We leave this as Exercise 17 and present a clearer but slightly less efficient algorithm in Figure 4.8. This algorithm avoids endless computation by remembering which nonterminals have already been visited, as follows.

- $\text{First}(\alpha)$ is computed by invoking $\text{FIRST}(\alpha)$.
- Before any sets are computed, Step 8 resets $\text{VisitedFirst}(A)$ for each nonterminal A .
- $\text{VisitedFirst}(\mathcal{X})$ is set at Step 12 to indicate that the productions of A already participate in the computation of $\text{First}(\alpha)$.

The primary computation is carried out by the function INTERNALFIRST , whose input argument is the string $\mathcal{X}\beta$. If $\mathcal{X}\beta$ is not empty, then \mathcal{X} is the string's first symbol and β is the rest of the string. INTERNALFIRST then computes its answer as follows.

- The empty set is returned if $\mathcal{X}\beta$ is empty at Step 9. We denote this condition by \perp to emphasize that the empty set is represented by a null list of symbols.
- If \mathcal{X} is a terminal, then $\text{First}(\mathcal{X}\beta)$ is $\{\mathcal{X}\}$ at Step 10.
- The only remaining possibility is that \mathcal{X} is a nonterminal. If $\text{VisitedFirst}(\mathcal{X})$ is false, then the productions for \mathcal{X} are recursively examined for inclusion. Otherwise, \mathcal{X} 's productions already participate in the current computation.
- If \mathcal{X} can derive λ at Step 14—this fact has been previously computed by the algorithm in Figure 4.7—then we must include all symbols in $\text{First}(\beta)$.

Level	First \mathcal{X}	β	ans	Step	Done?	Comment
COMPUTEFIRST(Tail)						
0	Tail	\perp	{ }	Step 11		
1	+	E	{ + }	Step 10	*	Tail \rightarrow +E
1	\perp	\perp	{ }	Step 9	*	Tail \rightarrow λ
0			{ + }	Step 13		After all rules for Tail
1	\perp	\perp	{ }	Step 9	*	Since $\beta = \perp$
0			{ + }	Step 14	*	Final answer
COMPUTEFIRST(Prefix)						
0	Prefix	\perp	{ }	Step 11		
1	f	\perp	{ f }	Step 10	*	Prefix \rightarrow f
1	\perp	\perp	{ }	Step 9	*	Prefix \rightarrow λ
0			{ f }	Step 13		After all rules for Prefix
1	\perp	\perp	{ }	Step 9	*	Since $\beta = \perp$
0			{ f }	Step 14	*	Final answer
COMPUTEFIRST(E)						
0	E	\perp	{ }	Step 11		
1	Prefix	(E)	{ }	Step 11		E \rightarrow Prefix (E)
1			{ f }	Step 15		Computation shown above
2	(E)	{ (}	Step 10	*	Since Prefix $\Rightarrow^* \lambda$
1			{ f, (}	Step 14	*	Results due to E \rightarrow Prefix (E)
1	v	Tail	{ v }	Step 10	*	E \rightarrow v Tail
1	\perp	\perp	{ }	Step 9		Since $\beta = \perp$
0			{ f, (, v }	Step 14	*	Final answer

Figure 4.9: First sets for the nonterminals of Figure 4.1.

Figure 4.9 shows the progress of COMPUTEFIRST as it is invoked on the nonterminals of Figure 4.1. The level of recursion is shown in the leftmost column. Each call to FIRST($\mathcal{X}\beta$) is shown with nonblank entries in the \mathcal{X} and β columns. A “*” indicates that the call does not recurse further. Figure 4.10 shows another grammar and the computation of its First sets; for brevity, recursive calls to INTERNALFIRST on null strings are omitted.

4.5.4 Follow Sets

Parser-construction algorithms often require the computation of the set of terminals that can follow a nonterminal A in some sentential form. Because we augment grammars to contain an end-of-input token ($\$$), every nonterminal ex-

$$\begin{array}{lcl}
 1 & S & \rightarrow A B c \\
 2 & A & \rightarrow a \\
 3 & & | \lambda \\
 4 & B & \rightarrow b \\
 5 & & | \lambda
 \end{array}$$

Level	First \mathcal{X}	β	<i>ans</i>	Step	Done?	Comment
COMPUTEFIRST(B)						
0	B	\perp	{ }	Step 11		
1	b	\perp	{ b }	Step 10	*	B \rightarrow b
1	\perp	\perp	{ }	Step 9	*	B \rightarrow λ
0			{ b }	Step 14	*	Final answer
COMPUTEFIRST(A)						
0	A	\perp	{ }	Step 11		
1	a	\perp	{ a }	Step 10	*	A \rightarrow a
1	\perp	\perp	{ }	Step 9	*	A \rightarrow λ
0			{ a }	Step 14	*	Final answer
COMPUTEFIRST(S)						
0	S	\perp	{ }	Step 11		
1	A	B c	{ a }	Step 15		Computation shown above
2	B	c	{ b }	Step 15		Because A \Rightarrow^* λ ; computation shown above
3	c	\perp	{ c }	Step 10	*	Because B \Rightarrow^* λ
2			{ b,c }	Step 14	*	
1			{ a,b,c }	Step 14	*	
0			{ a,b,c }	Step 14	*	

Figure 4.10: A grammar and its First sets.

```

function FOLLOW(A) : Set
  foreach A ∈ NONTERMINALS( ) do
    VisitedFollow(A) ← false 16
    ans ← INTERNALFOLLOW(A)
    return (ans)
end
function INTERNALFOLLOW(A) : Set
  ans ← ∅ 17
  if not VisitedFollow(A) 17
  then
    VisitedFollow(A) ← true 18
    foreach a ∈ OCCURRENCES(A) do 19
      ans ← ans ∪ FIRST(TAIL(a)) 20
      if ALLDERIVEEMPTY(TAIL(a)) 21
      then
        targ ← LHS(PRODUCTION(a))
        ans ← ans ∪ INTERNALFOLLOW(targ) 22
      return (ans) 23
    end
function ALLDERIVEEMPTY( $\gamma$ ) : Boolean
  foreach  $\mathcal{X} \in \gamma$  do
    if not SymbolDerivesEmpty( $\mathcal{X}$ ) or  $\mathcal{X} \in \Sigma$ 
    then return (false)
  return (true)
end

```

Figure 4.11: Algorithm for computing Follow(A).

cept the goal symbol *must* be followed by some terminal. Formally, for $A \in N$,

$$\text{Follow}(A) = \{ b \in \Sigma \mid S \Rightarrow^+ \alpha A b \beta \}.$$

Follow(A) provides the **right context** associated with nonterminal A. For example, only those terminals in Follow(A) can occur after a production for A is applied.

The algorithm shown in Figure 4.11 computes Follow(A). Many aspects of this algorithm are similar to the First(α) algorithm given in Figure 4.8.

- Follow(A) is computed by invoking FOLLOW(A).
- Before any sets are computed, Step 16 resets *VisitedFollow(A)* for each nonterminal A.
- *VisitedFollow(A)* is set at Step 18 to indicate that the symbols following A are already participating in this computation.

Level	Rule	Step	Result	Comment
COMPUTEFOLLOW(Prefix)				
0			FOLLOW(Prefix)	
0	$E \rightarrow \underline{\text{Prefix}} (E)$	Step 20	{ (}	
COMPUTEFOLLOW(E)				
0			FOLLOW(E)	
0	$E \rightarrow \text{Prefix} (\underline{E})$	Step 20	{) }	
0	$\text{Tail} \rightarrow + \underline{E}$	Step 22	{ }	
1			FOLLOW(Tail)	
1	$E \rightarrow v \underline{\text{Tail}}$	Step 22	{ }	
2			FOLLOW(E)	
		Step 17	{ }	Recursion avoided
1		Step 23	{ }	Returns
0		Step 23	{) }	Returns
COMPUTEFOLLOW(Tail)				
0			FOLLOW(Tail)	
0	$E \rightarrow v \underline{\text{Tail}}$	Step 22	{ }	
1			FOLLOW(E)	
1	$E \rightarrow \text{Prefix} (\underline{E})$	Step 20	{) }	
1	$\text{Tail} \rightarrow + \underline{E}$	Step 22	{ }	
2			FOLLOW(Tail)	
		Step 17	{ }	Recursion avoided
1		Step 23	{) }	Returns
0		Step 23	{) }	Returns

Figure 4.12: Follow sets for the nonterminals of Figure 4.1.

The primary computation is performed by INTERNALFOLLOW(A). Each occurrence a of A is visited by the loop at Step 19. TAIL(a) is the list of symbols immediately following the occurrence of A .

- Any symbol in First(TAIL(a)) can follow A . Step 20 includes such symbols in the returned set.
- Step 21 detects if the symbols in TAIL(a) can derive λ . This situation arises when there are no symbols appearing after this occurrence of A or when the symbols appearing after A can each derive λ . In either case, Step 22 includes the Follow set of the current production's LHS.

Figure 4.12 shows the progress of COMPUTEFOLLOW as it is invoked on the nonterminals of Figure 4.1. As another example, Figure 4.13 shows the computation of Follow sets for the grammar in Figure 4.10.

Level	Rule	Step	Result	Comment
			COMPUTEFOLLOW(B)	
0			FOLLOW(B)	
0	$S \rightarrow A \underline{B} c$	Step 20	{ c }	
0		Step 23	{ c }	Returns
			COMPUTEFOLLOW(A)	
0			FOLLOW(A)	
0	$S \rightarrow \underline{A} B c$	Step 20	{ b,c }	
0		Step 23	{ b,c }	Returns
			COMPUTEFOLLOW(S)	
0			FOLLOW(S)	
0		Step 23	{ }	Returns

Figure 4.13: Follow sets for the grammar in Figure 4.10. Note that $\text{Follow}(S) = \{ \}$ because S does not appear on the RHS of any production.

First and Follow sets can be generalized to include strings of length k rather than length 1. $\text{First}_k(\alpha)$ is the set of k -symbol terminal prefixes derivable from α . Similarly, $\text{Follow}_k(A)$ is the set of k -symbol terminal strings that can follow A in some sentential form. First_k and Follow_k are used in the definition of parsing techniques that use k -symbol lookaheads (for example, $\text{LL}(k)$ and $\text{LR}(k)$). The algorithms that compute $\text{First}_1(\alpha)$ and $\text{Follow}_1(A)$ can be generalized to compute $\text{First}_k(\alpha)$ and $\text{Follow}_k(A)$ sets (see Exercise 24).

This ends our discussion of CFGs and grammar-analysis algorithms. The First and Follow sets introduced in this chapter play an important role in the automatic construction of LL and LR parsers, as discussed in Chapters Chapter:global:five and Chapter:global:six, respectively.

Exercises

1. Transform the following grammar into a standard CFG using the algorithm in Figure 4.4.

1	S	→	Number
2	Number	→	[Sign] [Digs period] Digs
3	Sign	→	plus
4			minus
5	Digs	→	digit { digit }

2. Design a language and context-free grammar to represent the following languages.

- (a) The set of strings of base-8 numbers.
- (b) The set of strings of base-16 numbers.
- (c) The set of strings of base-1 numbers.
- (d) A language that offers base-8, base-16, and base-1 numbers.

3. Describe the language denoted by each of the following grammars.

- (a) $(\{A, B, C\}, \{a, b, c\}, \emptyset, A)$
- (b) $(\{A, B, C\}, \{a, b, c\}, \{A \rightarrow BC\}, A)$
- (c) $(\{A, B, C\}, \{a, b, c\}, \{A \rightarrow Aa, A \rightarrow b\}, A)$
- (d) $(\{A, B, C\}, \{a, b, c\}, \{A \rightarrow BB, B \rightarrow a, B \rightarrow b, B \rightarrow c\}, A)$

4. What are the difficulties associated with constructing a grammar whose generated strings are decimal representations of irrational numbers?

5. A grammar for infix expressions follows.

1	Start	→	E \$
2	E	→	T plus E
3			T
4	T	→	T times F
5			F
6	F	→	(E)
7			num

- (a) Show the leftmost derivation of the following string.

num plus num times num plus num \$

- (b) Show the rightmost derivation of the following string.

num times num plus num times num \$

- (c) Describe how this grammar structures expressions, in terms of the precedence and left- or right- associativity of operators.

6. Consider the following two grammars.

(a)

```

1 Start → E $
2 E     → ( E plus E
3       | num

```

(b)

```

1 Start → E $
2 E     → E ( plus E
3       | num

```

Which of these grammars, if any, is ambiguous? Prove your answer by showing two distinct derivations of some input string for the ambiguous grammar(s).

7. Compute First and Follow sets for the nonterminals of the following grammar.

```

1 S → a S e
2   | B
3 B → b B e
4   | C
5 C → c C e
6   | d

```

8. Compute First and Follow sets for each nonterminal in a grammar from Chapter 2, reprised as follows.

```

1 Prog → Dcls Stmts $
2 Dcls → Dcl Dcls
3       | λ
4 Dcl  → floatdcl id
5       | intdcl id
6 Stmts → Stmt Stmts
7       | λ
8 Stmt  → id assign Val ExprTail
9       | print id
10 ExprTail → plus Val ExprTail
11         | minus Val ExprTail
12         | λ
13 Val     → id
14         | num

```

9. Compute First and Follow sets for each nonterminal in Exercise 1.

10. As discussed in Section 4.3, the algorithm in Figure 4.4 could use left- or right-recursion to transform a repeated sequence of symbols into standard grammar form. A production of the form $A \rightarrow A \alpha$ is said to be **left recursive**. Similarly, a production of the form $A \rightarrow \beta A$ is said to be **right recursive**. Show that any grammar that contains left- and right-recursive rules for the same LHS nonterminal must be ambiguous.
11. Section 4.3 describes extended BNF notation for optional and repeated symbol sequences. Suppose the n grammar symbols $\mathcal{X}_1 \dots \mathcal{X}_n$ represent a set of n options. What is the effect of the following grammar with regard to how the options can appear?

$$\begin{array}{lcl} \text{Options} & \rightarrow & \text{Options Option} \\ & | & \lambda \\ \text{Option} & \rightarrow & \mathcal{X}_1 \\ & | & \mathcal{X}_2 \\ & & \dots \\ & | & \mathcal{X}_n \end{array}$$

12. Consider n optional symbols $\mathcal{X}_1 \dots \mathcal{X}_n$ as described in Exercise 11.
- Devise a CFG that generates any subset of these options. That is, the symbols can occur in any order, any symbol can be missing, and no symbol is repeated.
 - What is the relation between the size of your grammar and n , the number of options?
 - How is your solution affected if symbols \mathcal{X}_i and \mathcal{X}_j are present only if $i < j$?
13. Show that regular grammars and finite automata have equivalent definitional power by developing
- an algorithm that translates regular grammars into finite automata and
 - an algorithm that translates finite automata into regular grammars.
14. Devise an algorithm to detect nonterminals that cannot be reached from a CFG's goal symbol.
15. Devise an algorithm to detect nonterminals that cannot derive any terminal string in a CFG.
16. A CFG is reduced by removing useless terminals and productions. Consider the following two tasks.
- Nonterminals not reachable from the grammar's goal symbol are removed.
 - Nonterminals that derive no terminal string are removed.

Does the order of the above tasks matter? If so, which order is preferred?

17. The algorithm presented in Figure 4.8 retains no information between invocations of FIRST. As a result, the solution for a given nonterminal might be computed multiple times.
 - (a) Modify the algorithm so it remembers and references valid previous computations of $\text{First}(A)$, $A \in N$
 - (b) Frequently an algorithm needs First sets computed for *all* $X \in N$. Devise an algorithm that efficiently computes First sets for all nonterminals in a grammar. Analyze the efficiency of your algorithm.
Hint: Consider constructing a directed graph whose vertices represent nonterminals. Let an edge (A, B) represent that $\text{First}(B)$ depends on $\text{First}(A)$.
 - (c) Repeat this exercise for the Follow sets.
18. Prove that $\text{COMPUTEFIRST}(A)$ correctly computes $\text{First}(A)$ for any $A \in N$.
19. Prove that $\text{COMPUTEFOLLOW}(A)$ correctly computes $\text{Follow}(A)$ for any $A \in N$.
20. Let G be any CFG and assume $\lambda \notin L(G)$. Show that G can be transformed into a language-equivalent CFG that uses no λ -productions.
21. A **unit production** is a rule of the form $A \rightarrow B$. Show that any CFG that contains unit productions can be transformed into a language-equivalent CFG that uses no unit productions.
22. Some CFGs denote a language with an infinite number of strings; others denote finite languages. Devise an algorithm that determines whether a given CFG generates an infinite language.
Hint: Use the results of Exercises 20 and 21 to simplify the analysis.
23. Let G be an unambiguous CFG without λ -productions.
 - (a) If $x \in L(G)$, show that the number of steps needed to derive x is linear in the length of x .
 - (b) Does this linearity result hold if λ -productions are included?
 - (c) Does this linearity result hold if G is ambiguous?
24. The algorithms in Figures 4.8 and 4.11 compute $\text{First}(\alpha)$ and $\text{Follow}(A)$.
 - (a) Modify the algorithm in Figure 4.8 to compute $\text{First}_k(\alpha)$.
Hint: Consider formulating the algorithm so that when $\text{First}_i(\alpha)$ is computed, enough information is retained to compute $\text{First}_{i+1}(\alpha)$.
 - (b) Modify the algorithm in Figure 4.11 to compute $\text{Follow}_k(A)$.

Bibliography

- [HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [Mar91] John C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, 1991.

5

LL Grammars and Parsers

Chapter Chapter:global:two presents a *recursive-descent parser* for the syntax-analysis phase of a small compiler. Manual construction of such parsers is both time consuming and error prone—especially when applied at the scale of a real programming language. At first glance, the code for a recursive-descent parser may appear to be written *ad hoc*. Fortunately, there *are* principles at work. This chapter discusses these principles and their application in tools that automate the parsing phase of a compiler.

Recursive-descent parsers belong to the more general class of LL parsers, which were introduced in Chapter Chapter:global:four. In this chapter, we discuss LL parsers in greater detail, analyzing the conditions under which such parsers can be reliably and automatically constructed from **context-free grammars** (CFGs). Our analysis builds on the algorithms and grammar-processing concepts presented in Chapter Chapter:global:four. Because of their simplicity, performance, and excellent error diagnostics, recursive-descent parsers have been constructed for most modern programming languages.

In Section 5.2, we identify a subset of CFGs known as the $LL(k)$ grammars. In Sections 5.3 and 5.4, we show how to construct recursive-descent and table-driven LL parsers from $LL(1)$ grammars—an efficient subset of the $LL(k)$ grammars. For grammars that are not $LL(1)$, Section 5.5 considers grammar transformations that can eliminate non- $LL(1)$ properties. Unfortunately, some languages have no $LL(k)$ grammar, as discussed in Section 5.6. Section 5.7 establishes some useful properties of LL grammars and parsers. Parse-table representations are considered in Section 5.8. Because parsers are typically responsible for discovering syntax errors in programs, Section 5.9 considers how an $LL(k)$ parser might respond to syntactically faulty inputs.

5.1 Overview

In this chapter, we study the following two forms of LL parsers.

- **recursive-descent parsers** contain a set of mutually recursive procedures that cooperate to parse a string. The appropriate code for these procedures is determined by the particular $LL(k)$ grammar.
- **table-driven LL parsers** use a generic $LL(k)$ parsing engine and a parse table that directs the activity of the engine. The entries for the parse table are determined by the particular $LL(k)$ grammar.

Fortunately, CFGs with certain properties can be used to generate such parsers automatically. Tools that operate in this fashion are generally called **compiler-compilers** or **parser generators**. They take a grammar description file as input and attempt to produce a parser for the language defined by the grammar. The term “compiler-compiler” applies because the parser generator is itself a compiler—it accepts a high-level expression of a program (the grammar definition file) and generates an executable form of that program (the parser). This approach makes parsing one of the easiest and most reliable phases of compiler construction for the following reasons.

- When the grammar serves as a language's definition, parsers can be automatically constructed to perform syntax analysis in a compiler. The rigor of the automatic construction guarantees that the resulting parser is faithful to the language's syntactic specification.
- When a language is revised, updated, or extended, the associated modifications can be applied to the grammar description to generate a parser for the new language.
- When parser construction is successful through the techniques described in this chapter, the grammar is proved unambiguous. While devising an algorithmic test for grammar ambiguity is impossible, parsing techniques such as $LL(k)$ are of great use to language designers in developing intuition as to why a grammar might be ambiguous.

As discussed in Chapters Chapter:global:two and Chapter:global:four, every string in a grammar's language can be generated by a derivation that begins with the grammar's start symbol. While it is relatively straightforward to use a grammar's productions to generate sample strings in its language, reversing this process does not seem as simple. That is, given an input string, how can we show why the string is or is not in the grammar's language? This is the **parsing problem**, and in this chapter, we consider a parsing technique that is successful with many CFGs. This parsing technique is known by the following names:

- Top-down, because the parser begins with the grammar's start symbol and grows a parse tree from its root to its leaves

- Predictive, because the parser must predict at each step in the derivation which grammar rule is to be applied next
- LL(k), because these techniques scan the input from left-to-right, produce a leftmost derivation, and use k symbols of lookahead
- Recursive-descent, because this kind of parser can be implemented by a collection of mutually recursive procedures

5.2 LL(k) Grammars

Following is a reprise from Chapter Chapter:global:two of the process for constructing a recursive-descent parser from a CFG.

- A **parsing procedure** is associated with each nonterminal A .
- The procedure associated with A is charged with accomplishing one step of a derivation by choosing and applying one of A 's productions.
- The parser chooses the appropriate production for A by inspecting the next k tokens (terminal symbols) in the input stream. The **Predict set** for production $A \rightarrow \alpha$ is the set of tokens that trigger application of that production.
- The Predict set for $A \rightarrow \alpha$ is determined primarily by the detail in α —the **right-hand side** (RHS) of the production. Other CFG productions may participate in the computation of a production's Predict set.

Generally, the choice of production can be predicated on the next k tokens of input, for some constant k chosen before the parser is pressed into service. These k tokens are called the **lookahead** of an LL(k) parser. If it is possible to construct an LL(k) parser for a CFG such that the parser recognizes the CFG's language, then the CFG is an **LL(k) grammar**.

An LL(k) parser can peek at the next k tokens to decide which production to apply. However, the *strategy* for choosing productions must be established when the parser is constructed. In this section, we formalize this strategy by defining a function called $\text{Predict}_k(p)$. This function considers the grammar production p and computes the set of length- k token strings that predict the application of rule p . We assume henceforth that we have one token of lookahead ($k = 1$). We leave the generalization as Exercise 18. Thus, for rule p , $\text{Predict}(p)$ is the set of terminal symbols that call for applying rule p .

Consider a parser that is presented with the input string $\alpha a \beta \in \Sigma^*$. Suppose the parser has constructed the derivation $S \Rightarrow_{\text{lm}}^* \alpha A \mathcal{Y}_1 \dots \mathcal{Y}_n$. At this point, α has been matched and A is the leftmost nonterminal in the derived sentential form. Thus *some* production for A must be applied to continue the leftmost derivation. Because the input string contains an a as the next input token, the parse must continue with a production for A that derives a as its first terminal symbol.


```

function Predict( $p : A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_m$ ) : Set
   $ans \leftarrow \text{First}(\mathcal{X}_1 \dots \mathcal{X}_m)$                                 1
  if RuleDerivesEmpty( $p$ )                                       2
  then
     $ans \leftarrow ans \cup \text{Follow}(A)$                           3
  return ( $ans$ )
end

```

Figure 5.1: Computation of Predict sets.

Recalling the notation from Section Subsection:four:gramrep, we must examine the set of productions

$$P = \{ p \in \text{PRODUCTIONSFOR}(A) \mid a \in \text{Predict}(p) \}.$$

One of the following conditions must be true of the set P .

- P is the empty set. In this case, no production for A can cause the next input token to be matched. The parse cannot continue and a syntax error is issued, with a as the offending token. The productions for A can be helpful in issuing error messages that indicate which terminal symbols could be processed at this point in the parse. Section 5.9 considers error recovery and repair in greater detail.
- P contains more than one production. In this case, the parse could continue, but **nondeterminism** would be required to pursue the independent application of each production in P . For efficiency, we require that our parsers operate deterministically. Thus parser construction must ensure that this case cannot arise.
- P contains exactly one production. In this case, the leftmost parse can proceed deterministically by applying the only production in set P .

Fortunately, Predict sets are based on a grammar and not on any particular input string. We can analyze a grammar to determine whether each terminal predicts (at most) one of A 's rules. In such cases, we can construct a deterministic parser and we call the associated grammar LL(1).

We next consider a rule p in greater detail and show how to compute $\text{Predict}(p)$. Consider a production $p : A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_m, m \geq 0$.¹ As shown in Figure 5.1, the set of symbols that predict rule p is drawn from one or both of the following:

- The set of possible terminal symbols that are first produced in some derivation from $\mathcal{X}_1 \dots \mathcal{X}_m$
- Those terminal symbols that can follow A in some sentential form

¹Recall that by convention, if $m = 0$, then we have the rule $A \rightarrow \lambda$.

1	S	→	A C \$
2	C	→	c
3			λ
4	A	→	a B C d
5			B Q
6	B	→	b B
7			λ
8	Q	→	q
9			λ

Figure 5.2: A CFG.

In the algorithm of Figure 5.1, Step 1 initializes the result to $\text{First}(\mathcal{X}_1 \dots \mathcal{X}_m)$ —the set of terminal symbols that can appear leftmost in any derivation of $\mathcal{X}_1 \dots \mathcal{X}_m$. The algorithm for computing this set is given in Figure Figure:four:computeFIRST. Step 2 detects when $\mathcal{X}_1 \dots \mathcal{X}_m \Rightarrow^* \lambda$, using the results of the algorithm presented in Figure Figure:four:deriveLambda. $\text{RuleDerivesEmpty}(p)$ is true if, and only if, production p can derive λ . In this case, Step 3 includes those symbols in $\text{Follow}(A)$, as computed by the algorithm in Figure Figure:four:computeFollow. Such symbols can follow A after $A \Rightarrow^* \lambda$. Thus, the function shown in Figure 5.1 computes the set of length-1 token strings that predict rule p . By convention, λ is *not* a terminal symbol, so it does not participate in any Predict set.

In an LL(1) grammar, the productions for each A must have disjoint predict sets, as computed with one symbol of lookahead. Experience indicates that creating an LL(1) grammar is possible for most programming languages. However, *not all* CFGs are LL(1). For such grammars, the following may apply.

- More lookahead may be needed, in which case the grammar is LL(k) for some constant $k > 1$.
- A more powerful parsing method may be required. Chapter Chapter:global:six describes such methods.
- The grammar may be *ambiguous*. Such grammars cannot be accommodated by *any* deterministic parsing method.

We now apply the algorithm in Figure 5.1 to the grammar shown in Figure 5.2. Figure 5.3 shows the Predict calculation; for each production of the form $A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_m$, $\text{First}(\mathcal{X}_1 \dots \mathcal{X}_m)$ is shown. The next column indicates whether $\mathcal{X}_1 \dots \mathcal{X}_m \Rightarrow^* \lambda$. The rightmost column shows $\text{Predict}(A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_m)$ —the set of symbols that predict the production $A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_m$. This set includes $\text{First}(\mathcal{X}_1 \dots \mathcal{X}_m)$, as well as $\text{Follow}(A)$ if $\mathcal{X}_1 \dots \mathcal{X}_m \Rightarrow^* \lambda$.

The algorithm shown in Figure 5.4 determines whether a grammar is LL(1), based on the grammar's Predict sets. The Predict sets for each nonterminal A are checked for intersection. If no two rules for A have any symbols in common, then

Rule Number	A	$\mathcal{X}_1 \dots \mathcal{X}_m$	$\text{First}(\mathcal{X}_1 \dots \mathcal{X}_m)$	Derives Empty?	$\text{Follow}(A)$	Answer
1	S	A C \$	a,b,q,c,\$	No		a,b,q,c,\$
2	C	c	c	No		c
3		λ		Yes	d,\$	d,\$
4	A	a B C d	a	No		a
5		B Q	b,q	Yes	c,\$	b,q,c,\$
6	B	b B	b	No		b
7		λ		Yes	q,c,d,\$	q,c,d,\$
8	Q	q	q	No		q
9		λ		Yes	c,\$	c,\$

Figure 5.3: Predict calculation for the grammar of Figure 5.2.

```

function IsLL1(G) : Boolean
  foreach A ∈ N do
    PredictSet ← ∅
    foreach p ∈ ProductionsFor(A) do
      if Predict(p) ∩ PredictSet ≠ ∅
        then return (false)
      PredictSet ← PredictSet ∪ Predict(p)
    return (true)
  end

```

Figure 5.4: Algorithm to determine if a grammar G is LL(1).

the grammar is LL(1). The grammar of Figure 5.2 passes this test, and is therefore LL(1).

5.3 Recursive-Descent LL(1) parsers

We are now prepared to generate the procedures of a recursive-descent parser. The parser's input is a sequence of tokens provided by the stream ts . We assume that ts offers the following methods.

- PEEK, which examines the next input token without advancing the input.
- ADVANCE, which advances the input by one token

The parsers we construct rely on the MATCH method shown in Figure 5.5. This method checks the token stream ts for the presence of a particular token.

To construct a recursive-descent parser for an LL(1) grammar, we write a separate procedure for each nonterminal A . If A has rules p_1, p_2, \dots, p_n , we formulate the procedure shown in Figure 5.6. The code constructed for each p_i is obtained

```

procedure MATCH(ts, token)
  if ts.PEEK( ) = token
  then call ts.ADVANCE( )
  else call ERROR( Expected token )
end

```

Figure 5.5: Utility for matching tokens in an input stream.

```

procedure A(ts)
  switch ( )
  case ts.PEEK( ) ∈ Predict(p1)
    /* Code for p1 */
  case ts.PEEK( ) ∈ Predict(p2)
    /* Code for p2 */
  /* . */
  /* . */
  /* . */
  case ts.PEEK( ) ∈ Predict(pn)
    /* Code for pn */
  case default
    /* Syntax error */
end

```

Figure 5.6: A typical recursive-descent procedure.

by scanning the RHS of rule p_i ($\mathcal{X}_1 \dots \mathcal{X}_m$) from left to right. As each symbol is visited, code is inserted into the parsing procedure. For productions of the form $A \rightarrow \lambda$, $m = 0$ so there are no symbols to visit; in such cases, the parsing procedure simply returns immediately.

In considering \mathcal{X}_i , there are two possible cases, as follows.

1. \mathcal{X}_i is a terminal symbol. In this case, a call to $\text{MATCH}(ts, \mathcal{X}_i)$ is placed in the parser to insist that \mathcal{X}_i is the next symbol in the token stream. If the token is successfully matched, the token stream is advanced. Otherwise, the input string cannot be in the grammar's language and an error message is issued.
2. \mathcal{X}_i is a nonterminal symbol. In this case, there is a procedure responsible for continuing the parse by choosing an appropriate production for \mathcal{X}_i . Thus, a call to $\mathcal{X}_i(ts)$ is placed in the parser.

Figure 5.7 shows the parsing procedures created for the LL(1) grammar shown in Figure 5.2. For presentation purposes, the default case—representing a syntax error—is not shown in the parsing procedures of Figure 5.7.

```
procedure S(ts)
  switch ()
    case ts.PEEK() ∈ { a, b, q, c, $ }
      call A()
      call C()
      call MATCH(ts, $)
    end
  end
procedure C(ts)
  switch ()
    case ts.PEEK() ∈ { c }
      call MATCH(ts, c)
    case ts.PEEK() ∈ { d, $ }
      return ()
    end
  end
procedure A(ts)
  switch ()
    case ts.PEEK() ∈ { a }
      call MATCH(ts, a)
      call B()
      call C()
      call MATCH(ts, d)
    case ts.PEEK() ∈ { b, q, c, $ }
      call B()
      call Q()
    end
  end
procedure B(ts)
  switch ()
    case ts.PEEK() ∈ { b }
      call MATCH(ts, b)
      call B()
    case ts.PEEK() ∈ { q, c, d, $ }
      return ()
    end
  end
procedure Q(ts)
  switch ()
    case ts.PEEK() ∈ { q }
      call MATCH(ts, q)
    case ts.PEEK() ∈ { c, $ }
      return ()
    end
  end
end
```

Figure 5.7: Recursive-Descent Code.

5.4 Table-Driven LL(1) Parsers

The task of creating recursive-descent parsers as presented in Section 5.3 is mechanical and can therefore be automated. However, the size of the parser's code grows with the size of the grammar. Moreover, the overhead of method calls and returns can be a source of inefficiency. In this section we examine how to construct a table-driven LL(1) parser. Actually, the parser itself is standard across all grammars; we need only to provide an adequate parse table.

To make the transition from explicit code to table-driven processing, we use a stack to simulate the actions performed by `MATCH` and by the calls to the nonterminals' procedures. In addition to the methods typically provided by a stack, we assume that the top-of-stack contents can be obtained nondestructively (without popping the stack) via the method `TOS()`.

In code form, the generic LL(1) parser is given in Figure 5.8. At each iteration of the loop at Step 5, the parser performs one of the following actions.

- If the top-of-stack is a terminal symbol, then `MATCH` is called. This method, defined in Figure 5.5, ensures that the next token of the input stream matches the top-of-stack symbol. If successful, the call to `MATCH` advances the token input stream. For the table-driven parser, the matching top-of-stack symbol is popped at Step 9.
- If the top-of-stack is a nonterminal symbol, then the appropriate production is determined at Step 10. If a valid production is found, then `APPLY` is called to replace the top-of-stack symbol with the RHS of production p . These symbols are pushed such that the resulting top-of-stack is the first symbol on p 's RHS.

The parse is complete when the end-of-input symbol is matched at Step 8.

Given a CFG that has passed the ISLL1 test in Figure 5.4, we next examine how to build its LL(1) parse table. The rows and columns of the parse table are labeled by the nonterminals and terminals of the CFG, respectively. The table—consulted at Step 10 in Figure 5.8—is indexed by the top-of-stack symbol (`TOS()`) and by the next input token (`ts.PEEK()`).

Each nonblank entry in a row is a production that has the row's nonterminal as its **left-hand side** (LHS) symbol. A production is typically represented by its rule number in the grammar. The table is used as follows.

1. The nonterminal symbol at the top-of-stack determines which row is chosen.
2. The next input token (*i.e.*, the **lookahead**) determines which column is chosen.

The resulting entry indicates which, if any, production of the CFG should be applied at this point in the parse.

For practical purposes, the nonterminals and terminals should be mapped to small integers to facilitate table lookup by (two-dimensional) array indexing. The

```

procedure LLPARSER(ts)
  call PUSH(S)
  accepted  $\leftarrow$  false
  while not accepted do
    if TOS( )  $\in$   $\Sigma$ 
      then
        call MATCH(ts, TOS( ))
        if TOS( ) = $
          then accepted  $\leftarrow$  true
          call POP( )
        else
          p  $\leftarrow$  LLtable[TOS( ), ts.PEEK( )]
          if p = 0
            then call ERROR(Syntax error—no production applicable)
            else call APPLY(p)
      end
  procedure APPLY(p :  $A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_m$ )
    call POP( )
    for i = m downto 1 do
      call PUSH( $\mathcal{X}_i$ )
  end

```

Figure 5.8: Generic LL(1) parser.

```

procedure FILLTABLE(LLtable)
  foreach A  $\in$  N do
    foreach a  $\in$   $\Sigma$  do LLtable[A][a]  $\leftarrow$  0
  foreach A  $\in$  N do
    foreach p  $\in$  ProductionsFor(A) do
      foreach a  $\in$  Predict(p) do LLtable[A][a]  $\leftarrow$  p
  end

```

Figure 5.9: Construction of an LL(1) parse table.

Nonterminal	Lookahead					
	a	b	c	d	q	\$
S	1	1	1		1	1
C			2	3		3
A	4	5	5		5	5
B		6	7	7	7	7
Q			9		8	9

Figure 5.10: LL(1) table. Error entries are not shown.

procedure for constructing the parse table is shown in Figure 5.9. Upon the procedure's completion, any entry marked 0 will represent a terminal symbol that does not predict any production for the associated nonterminal. Thus, if a 0 entry is accessed during parsing, the input string contains an error.

Using the grammar shown in Figure 5.2 and its associated Predict sets shown in Figure 5.3, we construct the LL(1) parse table shown in Figure 5.10. The table's contents are the rule numbers for productions as shown in Figure 5.2, with blanks rather than zeros to represent errors.

Finally, using the parse table shown in Figure 5.10, we trace the behavior of an LL(1) parser on the input string a b b d c \$ in Figure 5.11.

5.5 Obtaining LL(1) Grammars

It can be difficult for inexperienced compiler writers to create LL(1) grammars. This is because LL(1) requires a unique prediction for each combination of nonterminal and lookahead symbol. It is easy to write productions that violate this requirement.

Fortunately, most LL(1) prediction conflicts can be grouped into two categories: *common prefixes* and *left-recursion*. Simple grammar transformations that eliminate common prefixes and left-recursion are known, and these transformations allow us to obtain LL(1) form for most CFGs.

5.5.1 Common Prefixes

In this category of conflicts, two productions for the same nonterminal share a **common prefix** if the productions' RHSs begin with the same string of grammar symbols. For the grammar shown in Figure 5.12, both Stmt productions are predicted by the if token. Even if we allow greater lookahead, the else that distinguishes the two productions can lie arbitrarily far ahead in the input, because Expr and StmtList can each generate a terminal string larger than any constant k . The grammar in Figure 5.12 is therefore not LL(k) for any k .

Prediction conflicts caused by common prefixes can be remedied by the simple factoring transformation shown in Figure 5.13. At Step 11 in this algorithm, a

Parse Stack	Action	Remaining Input
S		abdbc\$
\$CA	Apply 1: $S \rightarrow AC$	abdbc\$
\$CdCBa	Apply 4: $A \rightarrow aBCd$	abdbc\$
\$CdCB	Match	abdbc\$
\$CdCB		bdbc\$
\$CdCBb	Apply 6: $B \rightarrow bB$	bdbc\$
\$CdCB	Match	bdbc\$
\$CdCB		dbc\$
\$CdCBb	Apply 6: $B \rightarrow bB$	dbc\$
\$CdCB	Match	dbc\$
\$CdCB		dc\$
\$CdC	Apply 7: $B \rightarrow \lambda$	dc\$
\$Cd	Apply 3: $C \rightarrow \lambda$	dc\$
\$Cd		dc\$
\$C	Match	dc\$
\$C		c\$
\$c	Apply 2: $C \rightarrow c$	c\$
\$c		c\$
\$	Match	c\$
\$		\$
	Accept	

Figure 5.11: Trace of an LL(1) parse. The stack is shown in the left column, with top-of-stack as the rightmost character. The input string is shown in the right column, processed from left-to-right.

1	Stmt	→	if Expr then StmtList endif
2			if Expr then StmtList else StmtList endif
3	StmtList	→	StmtList ; Stmt
4			Stmt
5	Expr	→	var + Expr
6			var

Figure 5.12: A grammar with common prefixes.

```

procedure FACTOR( )
  foreach A ∈ N do
    α ← LongestCommonPrefix(ProductionsFor(A))
    while |α| > 0 do
      V ← NewNonTerminal()
      Productions ← Productions ∪ { A → αV }
      foreach p ∈ ProductionsFor(A) | RHS(p) = αβp do           11
        Productions ← Productions − { p }
        Productions ← Productions ∪ { V → βp }
      α ← LongestCommonPrefix(ProductionsFor(A))
  end

```

Figure 5.13: Factoring common prefixes.

```

1 Stmt    → if Expr then StmtList V1
2 V1    → endif
3         | else StmtList endif
4 StmtList → StmtList ; Stmt
5         | Stmt
6 Expr     → var V2
7 V2    → + Expr
8         | λ

```

Figure 5.14: Factored version of the grammar in Figure 5.12.

production is identified whose RHS shares a common prefix α with other productions; the remainder of the RHS is denoted β_p for production p . With the common prefix factored and placed into a new production for A , each production sharing α is stripped of this common prefix. Applying the algorithm in Figure 5.13 to the grammar in Figure 5.12 produces the grammar in Figure 5.14.

5.5.2 Left-Recursion

A production is **left-recursive** if its LHS symbol is also the first symbol of its RHS. In Figure 5.14, the production $\text{StmtList} \rightarrow \text{StmtList} ; \text{Stmt}$ is left-recursive. We extend this definition to nonterminals, so a nonterminal is left-recursive if it is the LHS symbol of a left-recursive production.

Grammars with left-recursive productions can never be LL(1). To see this, assume that some lookahead symbol t predicts the application of the left-recursive production $A \rightarrow A\beta$. With recursive-descent parsing, the application of this production will cause procedure A to be invoked repeatedly, without advancing the input. With the state of the parse unchanged, this behavior will continue indefinitely. Similarly, with table-driven parsing, application of this production will repeatedly push

```

procedure ELIMINATELEFTRECURSION( )
  foreach  $A \in N$  do
    if  $\exists r \in \text{ProductionsFor}(A) \mid \text{RHS}(r) = A\alpha$ 
    then
       $X \leftarrow \text{NewNonTerminal}()$ 
       $Y \leftarrow \text{NewNonTerminal}()$ 
      foreach  $p \in \text{ProductionsFor}(A)$  do
        if  $p = r$ 
        then  $\text{Productions} \leftarrow \text{Productions} \cup \{A \rightarrow XY\}$ 
        else  $\text{Productions} \leftarrow \text{Productions} \cup \{X \rightarrow \text{RHS}(p)\}$ 
       $\text{Productions} \leftarrow \text{Productions} \cup \{Y \rightarrow \alpha Y, Y \rightarrow \lambda\}$ 
  end

```

Figure 5.15: Eliminating left-recursion.

$A\beta$ on the stack without advancing the input.

The algorithm shown in Figure 5.15 removes left-recursion from a factored grammar. Consider the following left-recursive rules.

$$\begin{array}{l} 1 \quad A \rightarrow A\alpha \\ 2 \quad \quad \quad | \beta \end{array}$$

Each time Rule 1 is applied, an α is generated. The recursion ends when Rule 2 prepends a β to the string of α symbols. Using the regular-expression notation developed in Chapter Chapter:global:three, the grammar generates $\beta\alpha^*$. The algorithm in Figure 5.15 obtains a grammar that also generates $\beta\alpha^*$. However, the β is generated *first*. The α symbols are then generated via right-recursion. Applying this algorithm to the grammar in Figure 5.14 results in the grammar shown in Figure 5.16. Since X appears as the LHS of only one production, X 's unique RHS can be automatically substituted for all uses of X . This allows Rules 4 and 5 to be replaced with $\text{StmtList} \rightarrow \text{Stmt } Y$.

The algorithms presented in Figures 5.13 and 5.15 typically succeed in obtaining an LL(1) grammar. However, some grammars require greater thought to obtain an LL(1) version; some of these are included as exercises at the end of this chapter. All grammars that include the \$ (end-of-input) symbol can be rewritten into a form in which all right-hand sides begin with a terminal symbol; this form is called **Greibach Normal Form** (GNF) (see Exercise 19). Once a grammar is in GNF, factoring of common prefixes is easy. Surprisingly, even this transformation does not guarantee that a grammar will be LL(1) (see Exercise 20). In fact, as we discuss in the next section, language constructs do exist that have no LL(1) grammar. Fortunately, such constructs are rare in practice and can be handled by modest extensions to the LL(1) parsing technique.

1	Stmt	→	if Expr then StmtList V ₁
2	V ₁	→	endif
3			else StmtList endif
4	StmtList	→	X Y
5	X	→	Stmt
6	Y	→	; Stmt Y
7			λ
8	Expr	→	var V ₂
9	V ₂	→	+ Expr
10			λ

Figure 5.16: LL(1) version of the grammar in Figure 5.14.

1	S	→	Stmt \$
2	Stmt	→	if expr then Stmt else Stmt
3			if expr then Stmt
4			other

Figure 5.17: Grammar for if-then-else.

5.6 A Non-LL(1) Language

Almost all common programming language constructs can be specified by LL(1) grammars. One notable exception, however, is the if-then-else construct present in programming languages such as Pascal and C. The if-then-else language defined in Figure 5.16 has a token that *closes* an if—the `endif`. For languages that lack this delimiter, the if-then-else construct is subject to the so-called **dangling else** problem: A sequence of nested conditionals can contain more thens than elses, which leaves open the correspondence of thens to elses. Programming languages resolve this issue by mandating that each else is matched to its closest, otherwise unmatched then.

We next show that no LL(k) parser can handle languages that embed the if-then-else construct shown in Figure 5.17. This grammar has common prefixes that can be removed by the algorithm in Figure 5.13, but this grammar has a more serious problem. As demonstrated by Exercises 12 and 15, the grammar in Figure 5.17 is *ambiguous* and is therefore not suitable for LL(k) parsing. An **ambiguous grammar** can produce (at least) two distinct parses for some string in the grammar's language. Ambiguity is considered in greater detail in Chapter 6.

We do not intend to use the grammar of Figure 5.17 for LL(k) parsing. Instead, we study the *language* of this grammar to show that *no* LL(k) grammar exists for this language. In studies of this kind, it is convenient to elide unnecessary detail to expose a language's problematic aspects. In the language defined by the grammar of Figure 5.17, we can, in effect, regard the “if expr then Stmt” portion as an

<pre> 1 S → [S CL 2 λ 3 CL →] 4 λ </pre> <p style="text-align: center;">(a)</p>		<pre> 1 S → [S 2 T 3 T → [T] 4 λ </pre> <p style="text-align: center;">(b)</p>
--	--	---

Figure 5.18: Attempts to create an LL(1) grammar for DBL.

opening bracket and the “else Stmt” part as an *optional closing bracket*. Thus, the language of Figure 5.17 is structurally equivalent to the *dangling bracket language*, or DBL, defined by

$$\text{DBL} = \{ [^i]^j \mid i \geq j \geq 0 \}.$$

We next show that DBL is not LL(k) for any k .

We can gain some insight into the problem by considering some grammars for DBL. Our first attempt is the grammar shown in Figure 5.18(a), in which CL generates an optional closing bracket. Superficially, the grammar appears to be LL(1), because it is free of left-recursion and common prefixes. However, the ambiguity present in the grammar of Figure 5.17 is retained this grammar. Any sentential form containing CL CL can generate the terminal] two ways, depending on which CL generates the] and which generates λ . Thus, the string [] has two distinct parses.

To resolve the ambiguity, we create a grammar that follows the Pascal and C rules: Each] is matched with the *nearest unmatched* [. This approach results in the grammar shown in Figure 5.18(b). This grammar generates zero or more unmatched opening brackets followed by zero or more pairs of matching brackets. In fact, this grammar is parsable using most bottom-up techniques (such as SLR(1), which is discussed in Chapter Chapter:global:six). While this grammar is factored and is not left-recursive, it is not LL(1) according to the definitions given previously. The following analysis explains why this grammar is not LL(k) for any k .

$$\begin{aligned}
 [&\in \text{Predict}(S \rightarrow [S) \\
 [&\in \text{Predict}(S \rightarrow T) \\
 [[&\in \text{Predict}_2(S \rightarrow [S) \\
 [[&\in \text{Predict}_2(S \rightarrow T) \\
 &\dots \\
 [^k &\in \text{Predict}_k(S \rightarrow [S) \\
 [^k &\in \text{Predict}_k(S \rightarrow T)
 \end{aligned}$$

In particular, seeing only open brackets, LL parsers cannot decide whether to predict a matched or an unmatched open bracket. This is where bottom-up parsers have an

1	S	→	Stmt \$
2	Stmt	→	if expr then Stmt V
3			other
4	V	→	else Stmt
5			λ

Nonterminal	Lookahead					
	if	expr	then	else	other	\$
S	1				1	
Stmt	2				3	
V				4,5		5

Figure 5.19: Ambiguous grammar for if-then-else and its LL(1) table. The ambiguity is resolved by favoring Rule 4 over Rule 5 in the boxed entry.

advantage: They can delay applying a production until an entire RHS is matched. Top-down methods cannot delay—they must predict a production based on the first (or first k) symbols derivable from a RHS. To parse languages containing if-then-else constructs, the ability to postpone segments of the parse is crucial.

Our analysis thus concludes that LL(1) parser generators cannot automatically create LL(1) parsers from a grammar that embeds the if-then-else construct. This shortcoming is commonly handled by providing an ambiguous grammar along with some special-case rules for resolving any nonunique predictions that arise.

Factoring the grammar in Figure 5.17 yields the ambiguous grammar and (correspondingly nondeterministic) parse table shown in Figure 5.19. As expected, the else symbol predicts multiple productions—Rules 4 and 5. Since the else should match the closest then, we resolve the conflict in favor of Rule 4. Favoring Rule 5 would defer consumption of the else. Moreover, the parse table entry for nonterminal V and terminal else is Rule 4's only legitimate chance to appear in the parse table. If this rule is absent from the parse table, then the resulting LL(1) parser could never match *any* else. We therefore insist that rule $V \rightarrow \text{else Stmt}$ be predicted for V when the lookahead is else. The parse table or recursive-descent code can be modified manually to achieve this effect. Some parser generators offer mechanisms for establishing priorities when conflicts arise.

5.7 Properties of LL(1) Parsers

We can establish the following useful properties for LL(1) parsers.

- A correct, leftmost parse is constructed.

This follows from the fact that LL(1) parsers simulate a leftmost derivation. Moreover, the algorithm in Figure 5.4 finds a CFG to be LL(1) only if the

Predict sets of a nonterminal's productions are disjoint. Thus, the LL(1) parser traces the unique, leftmost derivation of an accepted string.

- All grammars in the LL(1) class are unambiguous.

If a grammar is ambiguous, then some string has two or more distinct leftmost derivations. If we compare two such derivations, then there must be a nonterminal A for which at least two different productions could be applied to obtain the different derivations. In other words, with a lookahead token of x , a derivation could continue by applying $A \rightarrow \alpha$ or $A \rightarrow \beta$. It follows that $x \in \text{Predict}(A \rightarrow \alpha)$ and $x \in \text{Predict}(A \rightarrow \beta)$. Thus, the test at Step 4 in Figure 5.4 determines that such a grammar is not LL(1).

- All table-driven LL(1) parsers operate in linear time and space with respect to the length of the parsed input. (Exercise 16 examines whether recursive-descent parsers are equally efficient.)

Consider the number of actions that can be taken by an LL(1) parser when the token x is presented as lookahead. Some number of productions will be applied before x either is matched or is found to be in error.

- Suppose a grammar is λ -free. In this case, no production can be applied twice without advancing the input. Otherwise, the cycle involving the same production will continue to be applied indefinitely; this condition should have been reported as an error when the LL(1) parser was constructed.
- If the grammar does include λ , then the number of nonterminals that could pop from the stack because of the application of λ -rules is proportional to the length of the input. Exercise 17 explores this point in more detail.

Thus each input token induces a bounded number of parser actions. It follows that the parser operates in linear time.

The LL(1) parser consumes space for the lookahead buffer—of constant size—and for the parse stack. The stack grows and contracts during parsing. However, the maximum stack used during any parse is proportional to the length of the parsed input, for either of the following reasons.

- The stack grows only when a production is applied of the form $A \rightarrow \alpha$. As argued previously, no production could be applied twice without advancing the input and, correspondingly, decreasing the stack size. If we regard the number and size of a grammar's productions to be bounded by some constant, then each input token contributes to a constant increase in stack size.
- If the parser's stack grew superlinearly, then the the parser would require more than linear time just to push entries on the stack.

Row	Column				
	1	2	3	4	5
1	L			P	
2		Q			R
3			U		
4	W	X			
5		Y		Z	

Figure 5.20: A sparse table.

5.8 Parse-Table Representation

Most entries in an LL(1) parse table are zero, indicating an error in the parser's input. LL(1) parse tables tend to be sparsely populated because the Predict sets for most productions are small relative to the size of the grammar's terminal vocabulary. For example, an LL(1) parser was constructed for a subset of Ada, using a grammar that contained 70 terminals and 138 nonterminals. Of the 9660 potential LL(1) parse-table entries, only 629 (6.5%) allowed the parse to continue.

Given such statistics, it makes sense to view the blank entries as a **default**; we then strive to represent the **nondefault** entries efficiently. Generally, consider a two-dimensional parse table with N rows, M columns, and E nondefault entries. The parse table constructed in Section 5.4 occupies space proportional to $N \times M$. Especially when $E \ll N \times M$, our goal is to represent the parse table using space proportional to E . Although modern workstations are equipped with ample storage to handle LL(1) tables for any practical LL(1) grammar, most computers operate more efficiently when storage accesses exhibit greater locality. A smaller parse table loads faster and makes better use of high-speed storage. Thus, it is worthwhile to consider sparse representations for LL(1) parse tables. However, any increase in space efficiency must not adversely affect the efficiency of accessing the parse table. In this section, we consider strategies for decreasing the size of a parse table T , such as the table shown in Figure 5.20.

5.8.1 Compaction

We begin by considering **compaction** methods that convert a table T into a representation devoid of default entries. Such methods operate as follows.

1. The nondefault entries of T are stored in compacted form.
2. A mapping is provided from the index pair (i, j) to the set $E \cup \{ \text{default} \}$.
3. The LL(1) parser is modified. Wherever the parser accesses $T[i, j]$, the mapping is applied to (i, j) and the compacted form supplies the contents of $T[i, j]$.

Compact Entry	Table T		Compact Entry	Table T		Hashes		
Index	Contents	Row	Column	Index	Contents	Row	Column	to
0	L	1	1	0	R	2	5	$10 \equiv 0$
1	P	1	4	1	L	1	1	1
2	Q	2	2	2	Y	5	2	$10 \equiv 0$
3	R	2	5	3	Z	5	4	$20 \equiv 0$
4	U	3	3	4	P	1	4	4
5	W	4	1	5	Q	2	2	4
6	X	4	2	6	W	4	1	4
7	Y	5	2	7				
8	Z	5	4	8	X	4	2	8
				9	U	3	3	9

Binary Search

(a)

Hash

(b)

Figure 5.21: Compact versions of the table in Figure 5.20. Only the boxed information is stored in a compact table.

Binary Search

The compacted form can be achieved by listing the nondefault entries in order of their appearance in T , scanning from left-to-right, top-to-bottom. The resulting compact table is shown in Figure 5.21(a). If row r of the compact table contains the nondefault entry $T[i, j]$, then row r contains also i and j , which are necessary for key comparison when the table is searched. We save space if $3 \times E < N \times M$, assuming each table entry takes one unit of storage. Because the data is sorted by row and column, the compact table can be accessed by **binary search**. Given E nondefault entries, each access takes $O(\log(E))$ time.

Hash Table

The compact table shown in Figure 5.21(b) uses $|E| + 1$ slots and stores $T[i, j]$ at a location determined by **hashing** i and j , using the hash function

$$h(i, j) = (i \times j) \bmod (|E| + 1).$$

To create the compact table, we process the nondefault entries of T in any order. The nondefault entry at $T[i, j]$ is stored in the compact table at $h(i, j)$ if that position is unoccupied. Otherwise, we search forward in the table, storing $T[i, j]$ at the next available slot. This method of handling collisions in the compact table is called **linear resolution**. Because the compact table contains $|E| + 1$ slots, one slot is always free after all nondefault entries are hashed. The vacant slot avoids an infinite loop when searching the compact table for a default entry.

Hash performance can be improved by allocating more slots in the compact table and by choosing a hash function that results in fewer collisions. Because the

nondefault entries of T are known in advance, both goals can be achieved by using **perfect hashing** [?]. With this technique, each nondefault entry $T[i, j]$ maps to one of $|E|$ slots using the key (i, j) . A nondefault entry is detected when the perfect hash function returns a value greater than $|E|$.

5.8.2 Compression

Compaction reduces the storage requirements of a parse table by eliminating default entries. However, the indices of a nondefault entry must be stored in the compact table to facilitate nondefault entry lookup. As shown in Figure 5.21, a given row or column index can be repeated multiple times. We next examine a **compression** method that tries to eliminate such redundancy and take advantage of default entries.

The compression algorithm we study is called **double-offset indexing**. The algorithm, shown in Figure 5.22, operates as follows.

1. The algorithm initializes a vector V at Step 12. Although the vector could hold $N \times M$ entries, the final size of the vector is expected to be closer to $|E|$. The entries of V are initialized to the parse table's default value.
2. Step 13 considers the rows of T in an arbitrary order.
3. When row i is considered, a shift value for the row is computed by the FINDSHIFT method. The shift value, retained in $R[i]$, records the amount by which an index into row i is shifted to find its entry in vector V . Method FITS checks that, when shifted, row i fits into V without any collision with the nondefault entries already established in V .
4. The size of V is reduced at Step 14 by removing all default values at V 's high end.

To use the compressed tables, entry $T[i, j]$ is found by inspecting V at location $l = R[i] + j$. If the row recorded at $V.fromrow[l]$ is i , then the table entry at $V.entry[l]$ is the nondefault table entry from $T[i, j]$. Otherwise, $T[i, j]$ has the default value.

We illustrate the effectiveness of the algorithm in Figure 5.22 by applying it to the sparse table shown in Figure 5.20. Suppose the rows are considered in order 1, 2, 3, 4, 5. The resulting structures, shown in Figure 5.23, can be explained as follows.

1. This row cannot be negatively shifted because it has an entry in column 1. Thus, $R[1]$ is 0 and $V[1 \dots 5]$ represents row 1, with nondefault entries at index 1 and 4.
2. This row can merge into V without shifting, because its nondefault values (columns 2 and 5) can be accommodated at 2 and 5, respectively.
3. Similarly, row 3 can be accommodated by V without any shifting.

```

procedure COMPRESS( )
  for  $i = 1$  to  $N \times M$  do 12
     $V.entry[i] \leftarrow default$ 
  foreach  $row \in \{1, 2, \dots, N\}$  do 13
     $R[row] \leftarrow FINDSHIFT(row)$ 
    for  $j = 1$  to  $M$  do
      if  $T[row, j] \neq default$ 
      then
         $place \leftarrow R[row] + j$ 
         $V.entry[place] \leftarrow T[row, j]$ 
         $V.fromrow[place] \leftarrow row$ 
    call TRUNC(  $V$  ) 14
  end
function FINDSHIFT(  $row$  ) : Integer
  return  $\left( \min_{shift=-M+1}^{N \times M - M} FITS(row, shift) \right)$ 
end
function FITS(  $row, shift$  ) : Boolean
  for  $j = 1$  to  $M$  do
    if  $T[row, j] \neq default$  and not ROOMINV(  $shift + j$  ) 15
    then return ( false )
  return ( true )
end
function ROOMINV(  $where$  ) : Boolean
  if  $where \geq 1$ 
  then
    if  $V.entry[where] = default$ 
    then return ( true )
  return ( false )
end
procedure TRUNC(  $V$  )
  for  $i = N \times M$  downto  $1$  do
    if  $V.entry[i] \neq default$ 
    then
      /* Retain  $V[1 \dots i]$  */
    return ( )
end

```

Figure 5.22: Compression algorithm.

R		V		
Row	Shift	Index	Entry	From
i	$R[i]$			Row
1	0	1	L	1
2	0	2	Q	2
3	0	3	U	3
4	5	4	P	1
5	6	5	R	2
		6	W	4
		7	X	4
		8	Y	5
		9		
		10	Z	5

Figure 5.23: Compression of the table in Figure 5.20. Only the boxed information is actually stored in the compressed structures.

4. When this row is considered, the first slot of V that can accommodate its leftmost column is slot 6. Thus, $R[4] = 5$ and row 4's nondefault entries are placed at 6 and 7.
5. Finally, columns 2 and 4 of row 5 can be accommodated at 8 and 10, respectively. Thus, $R[5] = 6$.

As suggested by the pseudocode at Step 13, rows can be presented to FINDSHIFT in any order. However, the size of the resulting compressed table can depend on the order in which rows are considered. Exercises 22 and 23 explore this point further. In general, finding a row ordering that achieves maximum compression is an **NP-complete** problem. This means that the best-known algorithms for obtaining optimal compression would have to try all row permutations. However, compression heuristics work well in practice. When compression is applied to the Ada LL(1) parse table mentioned previously, the number of entries drops from 9660 to 660. This result is only 0.3% from the 629 nondefault entries in the original table.

5.9 Syntactic Error Recovery and Repair

A compiler should produce a useful set of diagnostic messages when presented with a faulty input. Thus, when a single error is detected, it is usually desirable to continue processing the input to detect additional errors. Generally, parsers can continue syntax analysis using one of the following approaches.

- With *error recovery*, the parser attempts to ignore the current error. The parser enters a configuration where it is able to continue processing the input.

- *Error repair* is more ambitious. The parser attempts to correct the syntactically faulty program by modifying the input to obtain an acceptable parse.

In this section, we explore each of these approaches in turn. We then examine error detection and recovery for LL(1) parsers.

5.9.1 Error Recover

With **error recovery**, we try to reset the parser so that the remaining input can be parsed. This process may involve modifying the parse stack and remaining input. Depending on the success of the recovery process, subsequent syntax analysis may be accurate. Unfortunately, it is more often the case that faulty error recovery causes errors to **cascade** throughout the remaining parse. For example, consider the C fragment `a=func c+d`). If error recovery continues the parse by predicting a Statement after the `func`, then another syntax error is found at the parenthesis. A single syntax error has been amplified by error recovery by issuing two error messages.

The primary measure of quality in an error-recovery process is how few false or cascaded errors it induces. Normally, semantic analysis and code generation are disabled upon error recovery because there is no intention to execute the code of a syntactically faulty program.

A simple form of error recovery is often called **panic mode**. In this approach, the parser skips input tokens until it finds a frequently occurring delimiter (*e.g.*, a semicolon). The parser then continues by expecting those nonterminals that derive strings that can follow the delimiter.

5.9.2 Error Repair

With **error repair**, the parser attempts to repair the syntactically faulty program by modifying the parsed or (more commonly) the unparsed portion of the program. The compiler does not presume to know or to suggest an appropriate revision of the faulty program. The purpose of error repair is to analyze the offending input more carefully so that better diagnostics can be issued.

Error-recovery and error-repair algorithms can exploit the fact that LL(1) parsers have the **correct-prefix** property: For each state entered by such parsers, there is a string of tokens that could result in a successful parse. Consider the input string $\alpha x \beta$, where token x causes an LL(1) parser to detect a syntax error. The correct-prefix property means that there is at least one string $\alpha \gamma \neq \alpha x \beta$ that can be accepted by the parser.

What can a parser do to repair the faulty input? The following options are possible:

- Modification of α
- Insertion of text δ to obtain $\alpha \delta x \beta$

1	S	→	V
2			W
3	V	→	v A b
4	W	→	w A c
5	A	→	λ

Figure 5.24: An LL(1) grammar.

- Deletion of x to obtain $\alpha \beta$

These options are not equally attractive. The correct-prefix property implies that α is at least a portion of a syntactically correct program. Thus, most error recovery methods do not modify α except in special situations. One notable case is **scope repair**, where nesting brackets may be inserted or deleted to match the corresponding brackets in $x \beta$.

Insertion of text must also be done carefully. In particular, error repair based on insertion must ensure that the repaired string will not continually grow so that parsing can never be completed. Some languages are **insert correctable**. For such languages, it is always possible to repair syntactic faults by insertion. Deletion is a drastic alternative to insertion, but it does have the advantage of making progress through the input.

5.9.3 Error Detection in LL(1) Parsers

The recursive-descent and table-driven LL(1) parsers constructed in this chapter are based on Predict sets. These sets are, in turn, based on First and Follow information that is computed globally for a grammar. In particular, recall that the production $A \rightarrow \lambda$ is predicted by the symbols in $\text{Follow}(A)$.

Suppose that A occurs in the productions $V \rightarrow v A b$ and $W \rightarrow w A c$, as shown in Figure 5.24. For this grammar, the production $A \rightarrow \lambda$ is predicted by the symbols in $\text{Follow}(A) = \{b, c\}$. Examining the grammar in greater detail, we see that the application of $A \rightarrow \lambda$ should be followed only by b if the derivation stems from V . However, if the derivation stems from W , then A should be followed only by c . As described in this chapter, LL(1) parsing cannot distinguish between contexts calling for application of $A \rightarrow \lambda$. If the next input token is b or c , the production $A \rightarrow \lambda$ is applied, even though the next input token may not be acceptable. If the wrong symbol is present, the error is detected later, when matching the symbol after A in $V \rightarrow v A b$ or $W \rightarrow w A c$. Exercise 25 considers how such errors can be caught sooner by **full LL(1) parsers**, which are more powerful than the **strong LL(1) parsers** defined in this chapter.

```

1  S  →  [ E ]
2     |  ( E )
3  E  →  a

```

```

procedure S(ts, termset)
  switch ( )
    case ts.PEEK( ) ∈ { [ ] }
      call MATCH( [ ] )
      call E(ts, termset ∪ { [ ] } )
      call MATCH( ] )
    case ts.PEEK( ) ∈ { ( ) }
      call MATCH( ( ) )
      call E(ts, termset ∪ { ( ) } )
      call MATCH( ) )
  end
procedure E(ts, termset)
  if ts.PEEK( ) = a
  then call MATCH(ts, a)
  else
    call ERROR( Expected an a )
    while ts.PEEK( ) ∉ termset do call ts.ADVANCE( )
  end

```

Figure 5.25: A grammar and its Wirth-style, error-recovering parser.

5.9.4 Error Recovery in LL(1) Parsers

The LR(1) parsers described in Chapter Chapter:global:six are formally more powerful than the LL(1) parsers. However, the continued popularity of LL(1) parsers can be attributed, in part, to their superior error diagnosis and error recovery. Because of the predictive nature of an LL(1), leftmost parse, the parser can easily extend the parsed portion of a faulty program into a syntactically valid program. When an error is detected, the parser can produce messages informing the programmer of what tokens were *expected* so that the parse could have continued.

A simple and uniform approach to error recovery in LL(1) parsers is discussed by Wirth [Wir76]. When applied to recursive-descent parsers, the parsing procedures described in Section 5.3 are augmented with an extra parameter that receives a set of terminal symbols. Consider the parsing procedure $A(ts, termset)$ associated with some nonterminal A . When A is called during operation of the recursive-descent parser, any symbol passed via *termset* can legitimately serve as the lookahead symbol when *this* instance of A returns. For example, consider the

grammar and Wirth-style parsing procedures shown in Figure 5.25. Error recovery is placed in E so that if an *a* is not found, the input is advanced until a symbol is found that *can* follow E. The set of symbols passed to E includes those symbols passed to S as well as a closing bracket (if called from Step 16) or a closing parenthesis (if called from Step 17). If E detects an error, the input is advanced until a symbol in *termset* is found. Because end-of-input can follow S, every *termset* includes \$. In the worst case, the input program is advanced until \$, at which point all pending parsing procedures can exit.

Summary

This concludes our study of LL parsers. Given an LL(1) grammar, it is easy to construct recursive-descent or table-driven LL(1) parsers. Grammars that are not LL(1) can often be converted to LL(1) form by eliminating left-recursion and by factoring common prefixes. Some programming language constructs are inherently non-LL(1). Intervention by the compiler writer can often resolve the conflicts that arise in such cases. Alternatively, more powerful parsing methods can be considered, such as those presented in Chapter Chapter:global:six.

Exercises

1. Show why the following grammar is or is not LL(1).

1	S	→	A B c
2	A	→	a
3			λ
4	B	→	b
5			λ

2. Show why the following grammar is or is not LL(1).

1	S	→	A b
2	A	→	a
3			B
4			λ
5	B	→	b
6			λ

3. Show why the following grammar is or is not LL(1).

1	S	→	A B B A
2	A	→	a
3			λ
4	B	→	b
5			λ

4. Show why the following grammar is or is not LL(1).

1	S	→	a S e
2			B
3	B	→	b B e
4			C
5	C	→	c C e
6			d

5. Construct the LL(1) parse table for the following grammar.

1	Expr	→	- Expr
2			(Expr)
3			Var ExprTail
4	ExprTail	→	- Expr
5			λ
6	Var	→	id VarTail
7	VarTail	→	(Expr)
8			λ

6. Trace the operation of an LL(1) parser for the grammar of Exercise 5 on the following input.

id - id ((id))

7. Transform the following grammar into LL(1) form, using the techniques presented in Section 5.5.

1	DeclList	→	DeclList ; Decl
2			Decl
3	Decl	→	IdList : Type
4	IdList	→	IdList , id
5			id
6	Type	→	ScalarType
7			array (ScalarTypeList) of Type
8	ScalarType	→	id
9			Bound .. Bound
10	Bound	→	Sign intconstant
11			id
12	Sign	→	+
13			-
14			λ
15	ScalarTypeList	→	ScalarTypeList , ScalarType
16			ScalarType

8. Run your solution to Exercise 7 through any LL(1) parser generator to verify that it is actually LL(1). How do you know that your solution generates the same language as the original grammar?
9. Show that every regular language can be defined by an LL(1) grammar.
10. A grammar is said to have *cycles* if it contains a nonterminal A such that $A \Rightarrow^+ A$. Show that an LL(1) grammar must not have cycles.
11. Construct an LL(2) parser for the following grammar.

1	Stmt	→	id ;
2			id (IdList) ;
3	IdList	→	id
4			id , IdList

12. Show the two distinct parse trees that can be constructed for

if expr then if expr then other else other

using the grammar given in Figure 5.17. For each parse tree, explain the correspondence of then and else.

13. In Section 5.7, it is established that LL(1) parsers operate in linear time. That is, when parsing an input, the parser requires *on average* only a constant-bounded amount of time per input token.

Is it ever the case that an LL(1) parser requires more than a constant-bounded amount of time to accept some particular symbol? In other words, can we bound by a constant the time interval between successive calls to the scanner to obtain the next token?

14. Design an algorithm that reads an LL(1) parse table and produces the corresponding recursive-descent parser.
15. An **ambiguous grammar** can produce two distinct parses for some string in the grammar's language. Explain why an ambiguous grammar is never LL(k) for any k , even if the grammar is free of common prefixes and left-recursion.
16. Section 5.7 argues that table-driven LL(1) parsers operate in linear time and space. Explain why this claim does or does not hold for recursive-descent LL(1) parsers.
17. Explain why the number of nonterminals that can pop from an LL(1) parse stack is not bounded by a grammar-specific constant.
18. Design an algorithm that computes Predict_k sets for a CFG.
19. As discussed in Section 5.5, a grammar is in **Greibach Normal Form** (GNF) if all productions are of the form $A \rightarrow a\alpha$, where a is a terminal symbol and α is a string of zero or more grammar (*i.e.*, terminal or nonterminal) symbols. Let G be a grammar that does not generate λ . Design an algorithm to transform G into GNF.
20. If we construct a GNF version of a grammar using the algorithm developed in Exercise 19, the resulting grammar is free of left-recursion. However, the resulting grammar can still have common prefixes that prevent it from being LL(1). If we apply the algorithm presented in Figure 5.13 of Section 5.5.1, the resulting grammar will be free of left-recursion and common prefixes. Show that the absence of common prefixes and left-recursion in an unambiguous grammar does not necessarily make a grammar LL(1).
21. Section 5.7 and Exercises 16 and 17 examine the efficiency of LL(1) parsers.
- Analyze the efficiency of *operating* a table-driven LL(k) parser, assuming an LL(k) table has already been constructed. Your answer should be formulated in terms of the length of the parsed input.
 - Analyze the efficiency of *constructing* an LL(k) parse table. Your answer should be formulated in terms of the size of the grammar—its vocabularies and productions.
 - Analyze the efficiency of operating a recursive-descent LL(k) parser.

22. Apply the table compression algorithm in Figure 5.22 to the table shown in Figure 5.20, presenting rows in the order 1, 5, 2, 4, 3. Compare the success of compression with the result presented in Figure 5.23.
23. Although table-compression is an NP-complete problem, explain why the following heuristic works well in practice.

Rows are considered in order of decreasing density of nondefault entries. (That is, rows with the greatest number of nondefault entries are considered first.)

Apply this heuristic to the table shown in Figure 5.20 and describe the results.

24. A sparse array can be represented as a vector of rows, with each row represented as a *list* of nondefault column entries. Thus, the nondefault entry at $T[i, j]$ would appear as an element of list $R[j]$. The element would contain both its column identification (j) and the nondefault entry ($T[i, j]$).
- (a) Express the table shown in Figure 5.20 using this format.
- (b) Compare the effectiveness of this representation with those given in Section 5.8. Consider both the savings in space and any increase or decrease in access time.
25. Section 5.9.3 contains an example where the production $A \rightarrow \lambda$ is applied using an invalid lookahead token. With Follow sets computed globally for a given grammar, the style of LL(1) parsing described in this chapter is known as **strong LL(1)**. A **full LL(1)** parser applies a production only if the next input token is valid. Given an algorithm for constructing full LL(1) parse tables.
- Hint:* If a grammar contains n occurrences of the nonterminal A , then consider *splitting* this nonterminal so that each occurrence is a unique symbol. Thus, A is split into A_1, A_2, \dots, A_n . Each new nonterminal has productions similar to A , but the context of each nonterminal can differ.

26. Consider the following grammar:

1	S	→	V
2			W
3	V	→	v A b
4	W	→	w A c
5	A	→	λ

Is this grammar LL(1)? Is the grammar *full* LL(1), as defined in Exercise 25?

27. Section 5.9.4 describes an error recovery method that relies on dynamically constructed sets of Follow symbols. Compare these sets with the Follow information computed for full LL(1) in Exercise 25.

Do not copy

Bibliography

- [Cic86] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23:17–19, 1986.
- [Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs*. Prentice-Hall, 1976.

6

LR Parsing

Because of its power and efficiency, LR parsers are commonly used for the syntax-checking phase of a compiler. In this chapter, we study the operation and construction of LR parsers. Tools for the automatic construction of such parsers are available for a variety of platforms. These parser generators are useful not only because they automatically construct tables that drive LR parsers, but also because they are powerful diagnostic tools for developing or modifying grammars.

The basic properties and actions of a generic LR parser are introduced in Sections 6.1 and 6.2. Section 6.3 presents the most basic table-construction method for LR parsers. Section 6.4 considers problems that prevent automatic LR parser construction. Sections 6.5 and 6.6 discuss table-building algorithms of increasing sophistication and power. Of particular interest is the LALR(1) technique, which is used in most LR parser generators. The formal definition of most modern programming languages includes an LALR(1) grammar to specify the language's syntax.

6.1 Introduction

In Chapter Chapter:global:five, we learned how to construct LL (top-down) parsers based on **context-free grammars** (CFGs) that had certain properties. The fundamental concern of a LL parser is which production to choose in expanding a given nonterminal. This choice is based on the parser's current state and on a peek at the unconsumed portion of the parser's input string. The derivations and parse trees produced by LL parsers are easy to follow: the leftmost nonterminal is expanded at each step, and the parse tree grows systematically—top-down, from left to right. The LL parser begins with the tree's root, which is labeled with the grammar's goal symbol. Suppose that A is the next nonterminal to be expanded, and that the parser

chooses the production $A \rightarrow \gamma$. In the parse tree, the node corresponding to this A is supplied with children that are labeled with the symbols in γ .

In this chapter, we study LR (bottom-up) parsers, which appear to operate in the reverse direction of LL parsers.

- The LR parser begins with the parse tree's leaves and moves toward its root.
- The *rightmost* derivation is produced *in reverse*.
- An LR parser uses a grammar rule to replace the rule's **right-hand side** (RHS) with its **left-hand side** (LHS).

Figures Figure:four:tdparse and Figure:four:buparse illustrate the differences between a top-down and bottom-up parse. Section 6.2 considers bottom-up parses in greater detail.

Unfortunately, the term “LR” denotes both the generic bottom-up parsing engine as well as a particular technique for building the engine's tables. Actually, the style of parsing considered in this chapter is known by the following names.

- **Bottom-up**, because the parser works its way from the terminal symbols to the grammar's goal symbol
- **Shift-reduce**, because the two most prevalent actions taken by the parser are to *shift* symbols onto the parse stack and to *reduce* a string of such symbols located at the top-of-stack to one of the grammar's nonterminals
- **LR(k)**, because such parsers scan the input from the left (L) producing a rightmost derivation (R) in reverse, using k symbols of lookahead

In an LL parser, each state is committed to expand a particular nonterminal. On the other hand, an LR parser can concurrently anticipate the eventual success of multiple nonterminals. This flexibility makes LR parsers more general than LL parsers. For example, LL parsers cannot accommodate left-recursive grammars and they cannot automatically handle the “dangling-else” problem described in Chapter Chapter:global:five. LR parsers have no systematic problem with either of these areas.

Tools for the automatic construction of LR parsers are available for a variety of platforms, including ML, Java, C, and C++. Chapter Chapter:global:seven discusses such tools in greater detail. It is important to note that a parser generator for a given platform performs syntax analysis for the language of its provided grammar. For example, the Yacc is a popular parser generator that emits C code. If Yacc is given a grammar for the syntax of FORTRAN, then the resulting parser compiles using C but performs syntax analysis for FORTRAN. The syntax of most modern programming languages is defined by grammars that are suitable for automatic parser generation using LR techniques.

1	Start	→	E \$
2	E	→	plus E E
3			num

Rule	Derivation
1	Start \Rightarrow_{rm} E \$
2	\Rightarrow_{rm} plus E E \$
3	\Rightarrow_{rm} plus E num \$
3	\Rightarrow_{rm} plus num num \$

Figure 6.1: Rightmost derivation of plus num num \$.

6.2 Shift-Reduce Parsers

In this section, we examine the operation of an LR parser, assuming that an LR parse table has already been constructed to guide the parser's actions. The reader may be understandably curious about how the table's entries are determined. However, table-construction techniques are best considered after obtaining a solid understanding of an LR parser's operation.

We describe the operation of an LR parser informally in Sections 6.2.1 and 6.2.2. Section 6.2.3 describes a generic LR parsing engine whose actions are guided by the LR parse table defined in Section 6.2.4. Section 6.2.5 presents LR(k) parsing more formally.

6.2.1 LR Parsers and Rightmost Derivations

One method of understanding an LR parse is to appreciate that such parses construct rightmost derivations in reverse. Given a grammar and a rightmost derivation of some string in its language, the sequence of productions applied by an LR parser is the sequence used by the rightmost derivation—played backwards. Figure 6.1 shows a grammar and the rightmost derivation of a string in the grammar's language. The language is suitable for expressing sums in a prefix (Lisp-like) notation. Each step of the derivation is annotated with the production number used at that step. For this example, the derivation uses Rules 1, 2, 3, and 3.

A bottom-up parse is accomplished by playing this sequence backwards—Rules 3, 3, 2, and 1. In contrast with LL parsing, an LR parser finds the RHS of a production and replaces it with the production's LHS. First, the leftmost num is **reduced** to an E by the rule $E \rightarrow \text{num}$. This rule is applied again to obtain plus E E \$. The sum is then reduced by $E \rightarrow \text{plus E E}$ to obtain E \$. This can then be reduced by Rule 1 to the goal symbol Start.

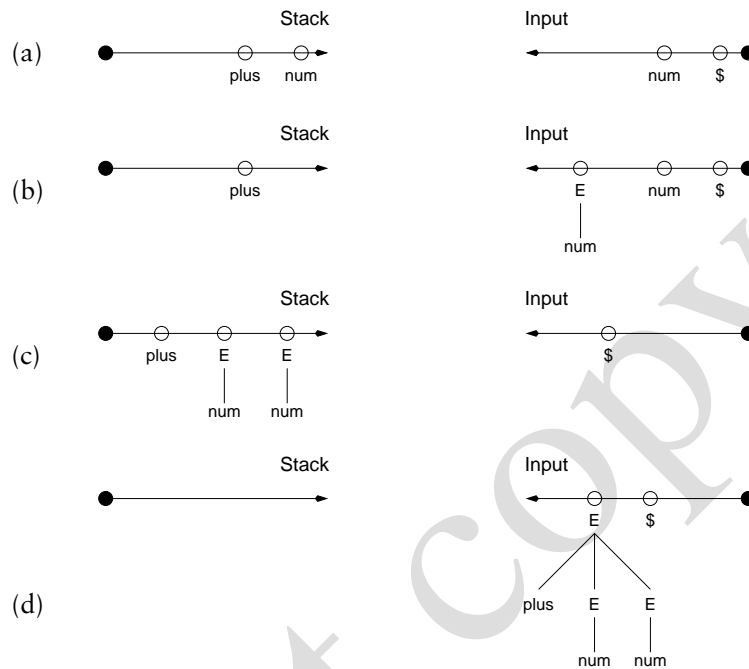


Figure 6.2: Bottom-up parsing resembles knitting.

6.2.2 LR Parsing as Knitting

Section 6.2.1 presents the order in which productions are applied to perform a bottom-up parse. We next examine *how* the RHS of a production is found so that a reduction can occur. The actions of an LR parser are analogous to *knitting*. Figure 6.2 illustrates this by showing a parse in progress for the grammar and string of Figure 6.1. The right needle contains the currently unprocessed portion of the string. The left needle is the parser's stack, which represents the processed portion of the input string.

A **shift** operation transfers a symbol from the right needle to the left needle. When a **reduction** by the rule $A \rightarrow \gamma$ is performed, the symbols in γ must occur at the pointed end of the left needle—at the *top* of the parse stack. Reduction by $A \rightarrow \gamma$ removes the symbols in γ and prepends the LHS symbol A to the unprocessed input of the right needle. A is then treated as an input symbol to be shifted onto the left needle. To illustrate the parse tree under construction, Figure 6.2 shows the symbols in γ as children of A .

We now follow the parse that is illustrated in Figure 6.2. In Figure 6.2(a), the left needle shows that two shifts have been performed. With plus num on the left needle, it is time to reduce by $E \rightarrow \text{num}$. Figure 6.2(b) shows the effect of this reduction, with the resulting E prepended to the input. This same sequence of

```

call Stack.PUSH(StartState)
accepted ← false
while not accepted do
  action ← Table[Stack.TOS( )][InputStream.PEEK( )]           1
  if action = shift s
  then
    call Stack.PUSH(s)                                       2
    if s ∈ AcceptStates                                     3
    then accepted ← true
    else call InputStream.ADVANCE( )
  else
    if action = reduce A → γ
    then
      call Stack.POP(|γ|)                                     4
      call InputStream.PREPEND(A)                             5
    else
      call ERROR( )                                         6

```

Figure 6.3: Driver for a bottom-up parser.

activities is repeated to obtain the state shown in Figure 6.2(c)—the left needle contains plus E E. When reduced by $E \rightarrow \text{plus } E E$, we obtain Figure 6.2(d). The resulting E \$ would then be shifted, reduced by $\text{Start} \rightarrow E \$$, and the parse would be accepted.

Based on the input string and the sequence of shift and reduce actions, symbols transfer back and forth between the needles. The “product” of the knitting is the parse tree, shown on the left needle if the input string is accepted.

6.2.3 LR Parsing Engine

Before considering an example in some detail, we present a simple driver for our shift-reduce parser in Figure 6.3. The parsing engine is driven by a table, whose entries are discussed in Section 6.2.4. The table is indexed at Step 1, using the parser's *current state* and the next (unprocessed) input symbol. The **current state** of the parser is defined by the contents of the parser's stack. To avoid rescanning the stack's contents prior to each parser action, state information is computed and stored with each symbol shifted onto the stack. Thus, Step 1 need only consult the state information associated with the stack's topmost symbol. The parse table calls for a shift or reduce as follows.

- Step 2 performs a shift of the next input symbol to state s .
- A reduction occurs at Steps 4 and 5. The RHS of a production is popped off the stack and its LHS symbol is prepended to the input.

The parser continues to perform shift and reduce actions until one of the following situations occurs.

- The input is reduced to the grammar's goal symbol at Step 3. The input string is **accepted**.
- No valid action is found at Step 1. In this case, the input string has a syntax error.

6.2.4 The LR Parse Table

We have seen that an LR parse constructs a rightmost derivation in reverse. Each reduction step in the LR parse uses a grammar rule such as $A \rightarrow \gamma$ to replace γ by A . A sequence of **sentential forms** is thus constructed, beginning with the input string and ending with the grammar's goal symbol.

Given a sentential form, the **handle** is defined as the sequence of symbols that will next be replaced by reduction. The difficulties lie in identifying the handle and in knowing which production to employ in the reduction (should there be multiple productions with the same RHS). These activities are choreographed by the parse table.

A suitable parse table for the grammar in Figure 6.4 is shown in Figure 6.5. This grammar appeared in Figure Figure:five:simplegram to illustrate top-down parsing. Readers familiar with top-down parsing can use this grammar to compare the methods.

To conserve space, a shift and reduce actions are distinguished graphically in our parse tables. A shift to State s is denoted by \boxed{s} . An unboxed number is the production number for a reduction. Blank entries are error actions, and the parser accepts when the Start symbol is shifted in the parser's starting state. Using the table in Figure 6.5, Figure 6.6 shows the steps of a bottom-up parse. For pedagogical purposes, each stack cell is shown as two elements: $\begin{matrix} a \\ n \end{matrix}$. The bottom element n is the parser state entered when the cell is pushed. The top symbol a is the symbol causing the cell to be pushed. The parsing engine described in Figure 6.3 keeps track only of the state.

The reader should verify that the reductions taken in Figure 6.6 trace a rightmost derivation in reverse. Moreover, the shift actions are essentially implied by the reductions: tokens are shifted until the handle appears at the top of the parse stack, at which time the next reduction in the reverse derivation can be applied.

Of course, the parse table plays a central role in determining the shifts and reductions that are necessary to recognize a valid string. For example, the rule $C \rightarrow \lambda$ could be applied at any time, but the parse table calls for this only in certain states and only when certain tokens are next in the input stream.

1	Start	→	S \$
2	S	→	A C
3	C	→	c
4			λ
5	A	→	a B C d
6			B Q
7	B	→	b B
8			λ
9	Q	→	q
10			λ

Rule	Derivation
1	Start \Rightarrow_{rm} S \$
2	\Rightarrow_{rm} A C \$
3	\Rightarrow_{rm} A c \$
5	\Rightarrow_{rm} a B C d c \$
4	\Rightarrow_{rm} a B d c \$
7	\Rightarrow_{rm} a b B d c \$
7	\Rightarrow_{rm} a b b B d c \$
8	\Rightarrow_{rm} a b b d c \$

Figure 6.4: Rightmost derivation of a b b d c \$.

6.2.5 LR(k) Parsing

The concept of LR parsing was introduced by Knuth [?]. As was the case with LL parsers, LR parsers are parameterized by the number of lookahead symbols that are consulted to determine the appropriate parser action: an LR(k) parser can peek at the next k tokens. This notion of “peeking” and the term LR(0) are confusing, because even an LR(0) parser must refer to the next input token, for the purpose of indexing the parse table to determine the appropriate action. The “0” in LR(0) refers not to the lookahead at parse-time, but rather to the lookahead used in *constructing* the parse table. At parse-time, LR(0) and LR(1) parsers index the parse table using one token of lookahead; for $k \geq 2$, an LR(k) parser uses k tokens of lookahead.

The number of columns in an LR(k) parse table grows dramatically with k . For example, an LR(3) parse table is indexed by the parse state to select a row, and by the next 3 input tokens to select a column. If the terminal alphabet has n symbols, then the number of distinct three-token sequences is n^3 . More generally, an LR(k) table has n^k columns for a token alphabet of size n . To keep the size of parse tables within reason, most parser generators are limited to one token of lookahead. Some parser generators do make selected use of extra lookahead where such information is helpful.

Most of this chapter is devoted to the problems of constructing LR parse tables.

State	a	b	c	d	q	\$	Start	S	A	B	C	Q
0	3	2	8		8	8	accept	4	1	5		
1			11			4					14	
2		2	8	8	8	8				13		
3		2	8	8						9		
4						8						
5			10		7	10						6
6			6			6						
7			9			9						
8						1						
9			11	4							10	
10				12								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar shown in Figure 6.4.

Before we consider such techniques, it is instructive to formalize the definition of $LR(k)$ in terms of the properties an $LR(k)$ parser must possess. All shift-reduce parsers operate by shifting symbols and examining lookahead information until the end of the handle is found. Then the handle is reduced to a nonterminal, which replaces the handle on the stack. An $LR(k)$ parser, guided by its parse table, must decide whether to shift or reduce, knowing only the symbols already shifted (left context) and the next k lookahead symbols (right context).

A grammar is $LR(k)$ if and only if it is possible to construct an LR parse table such that k tokens of lookahead allows the parser to recognize *exactly* those strings in the grammar's language. An important property of an LR parse table is that each cell accommodates only one entry. In other words, the $LR(k)$ parser is **deterministic**—exactly one action can occur at each step.

We next formalize the properties of an $LR(k)$ grammar, using the following definitions from Chapter Chapter:global:four.

- If $S \Rightarrow^* \beta$, then β is a **sentential form** of a grammar with goal symbol S .
- $\text{First}_k(\alpha)$ is the set of length- k terminal prefixes that can be derived from α .

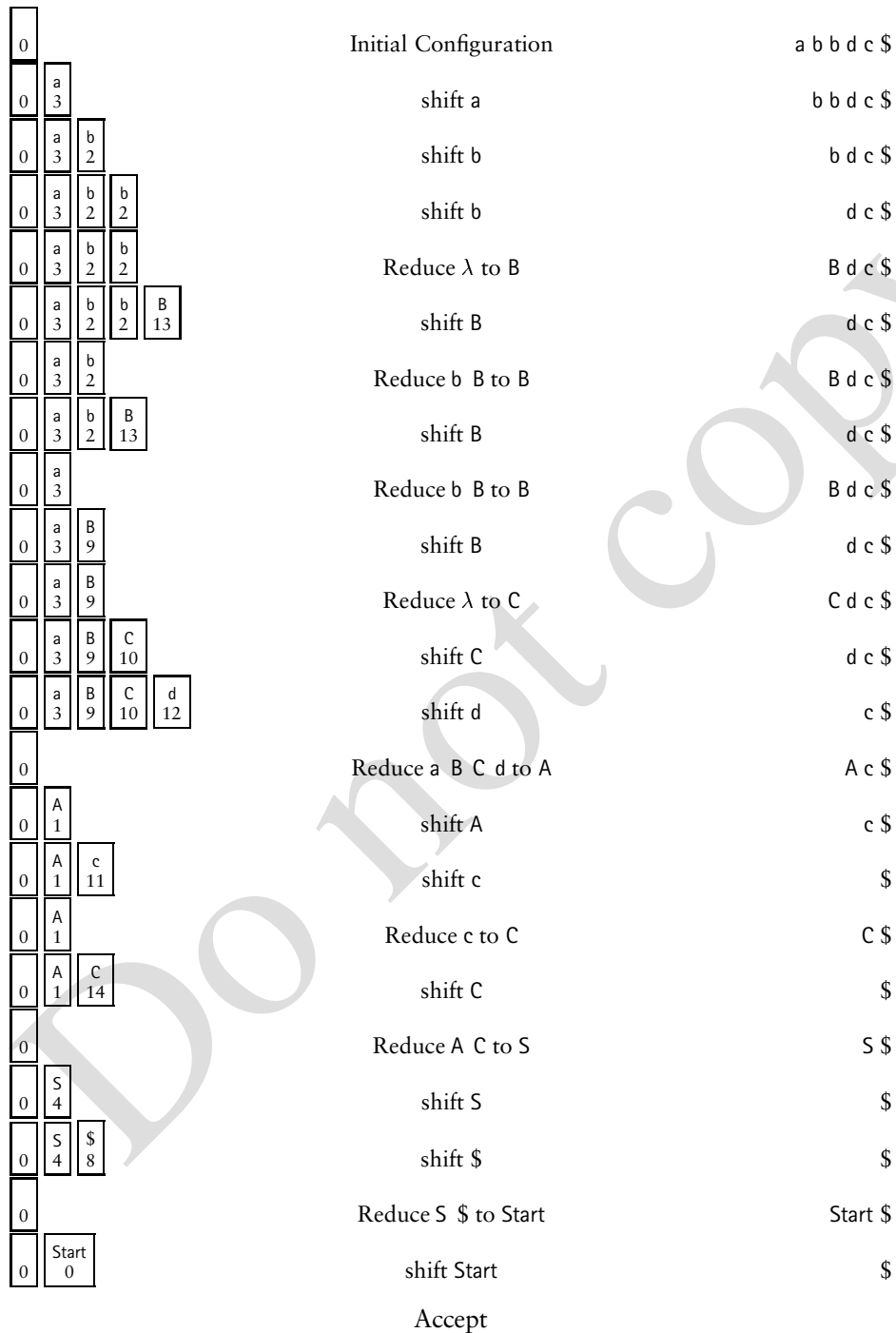


Figure 6.6: Bottom-up parse of a b b d c \$.

Assume that in some $LR(k)$ grammar there are two sentential forms $\alpha\beta w$ and $\alpha\beta y$, with $w, y \in \Sigma^*$. These sentential forms share a common prefix $\alpha\beta$. Further, with the prefix $\alpha\beta$ on the stack, their k -token lookahead sets are identical: $\text{First}_k(w) = \text{First}_k(y)$. Suppose the parse table calls for reduction by $A \rightarrow \beta$ given the left context of $\alpha\beta$ and the k -token lookahead present in w ; this results in αAw . With the same lookahead information in y , the $LR(k)$ parser insists on making the same decision: $\alpha\beta y$ becomes αAy . Formally, a grammar is $LR(k)$ if and only if the following conditions imply $\alpha Ay = \gamma Bx$.

1. $S \Rightarrow_{rm}^* \alpha Aw \Rightarrow_{rm} \alpha\beta w$
2. $S \Rightarrow_{rm}^* \gamma Bx \Rightarrow_{rm} \alpha\beta y$
3. $\text{First}_k(w) = \text{First}_k(y)$

This implication allows reduction by $A \rightarrow \beta$ whenever β is on top-of-stack and the k -symbol lookahead is $\text{First}_k(w)$.

This definition is instructive in that it defines the minimum properties a grammar must possess to be parsable by $LR(k)$ techniques. It does not tell us how to *build* a suitable $LR(k)$ parser; in fact, the primary contribution of Knuth's early work was an algorithm for $LR(k)$ construction. We begin with the simplest $LR(0)$ parser, which lacks sufficient power for most applications. After examining problems that arise in $LR(0)$ construction we turn to the more powerful $LR(1)$ parsing method and its variants.

When LR parser construction fails, the associated grammar may be *ambiguous* (discussed in Section 6.4.1). For other grammars, the parser may require more information about the unconsumed input string (lookahead). In fact, some grammars require an *unbounded* amount of lookahead (Section 6.4.2). In either case, the parser-generator identifies *inadequate* parsing states that are useful for resolving the problem. While it can be shown that there can be no algorithm to determine if a grammar is ambiguous, Section 6.4 describes techniques that work well in practice.

6.3 $LR(0)$ Table Construction

The table-construction methods discussed in this chapter analyze a grammar to devise a parse table suitable for use in the generic parser presented in Figure 6.3. Each symbol in the terminal and nonterminal alphabets corresponds to a column of the table. The analysis proceeds by exploring the state-space of the parser. Each state corresponds to a row of the parse table. Because the state-space is necessarily finite, this exploration must terminate at some point. After the parse table's rows have been determined, analysis then attempts to fill in the cells of the table. Because we are interested only in *deterministic* parsers, each table cell can hold one entry. An important outcome of the LR construction methods is the determination of **inadequate states**—states that lack sufficient information to place at most one parsing action in each column.

LR(0) item	Progress of rule in this state
$E \rightarrow \bullet \text{ plus } E E$	Beginning of rule
$E \rightarrow \text{ plus } \bullet E E$	Processed a plus, expect an E
$E \rightarrow \text{ plus } E \bullet E$	Expect another E
$E \rightarrow \text{ plus } E E \bullet$	Handle on top-of-stack, ready to reduce

Figure 6.7: LR(0) items for production $E \rightarrow \text{ plus } E E$.

We next consider LR(0) table construction for the grammar shown in Figure 6.1. In constructing the parse table, we are required to consider the parser's progress in recognizing a rule's **right-hand side** (RHS). For example, consider the rule $E \rightarrow \text{ plus } E E$. Prior to reducing the RHS to E, each component of the RHS must be found. A plus must be identified, then two Es must be found. Once these three symbols are on top-of-stack, then it is possible for the parser to apply the reduction and replace the three symbols with the **left-hand side** (LHS) symbol E.

To keep track of the parser's progress, we introduce the notion of an **LR(0) item**—a grammar production with a bookmark that indicating the current progress through the production's RHS. The bookmark is analogous to the “progress bar” present in many applications, indicating the completed fraction of a task. Figure 6.7 shows the possible LR(0) items for the production $E \rightarrow \text{ plus } E E$. A *fresh* item has its marker at the extreme left, as in $E \rightarrow \bullet \text{ plus } E E$. When the marker is at the extreme right, as in $E \rightarrow \text{ plus } E E \bullet$, we say the item is *reducible*. As a special case, consider the rule $A \rightarrow \lambda$. The symbol “ λ ” denotes that there is *nothing* on this rule's RHS. We make this clear when representing such rules as items. For $A \rightarrow \lambda$, the only possible item is the reducible $A \rightarrow \bullet$.

We now define a **parser state** as a set of LR(0) items. While each state is formally a set, we drop the usual braces notation and simply list the the set's elements (items). The LR(0) construction algorithm is shown in Figure 6.8.

The start state for our parser—nominally state 0—is formed at Step 7 by including fresh items for each of the grammar's goal-symbol productions. For our example grammar in Figure 6.1, we initialize the start state with $\text{Start} \rightarrow \bullet E \$$. The algorithm maintains *WorkList*—a set of states that need to be processed by the loop at Step 8. Each state formed during processing is passed through ADDSTATE, which determines at Step 9 if the set of items has already been identified as a state. If not, then a new state is constructed at Step 10. The resulting state is added to *WorkList* at Step 11. The state's row in the parse table is initialized at Step 12.

The processing of a state begins when the loop at Step 8 extracts a state s from the *WorkList*. When COMPUTEGOTO is called on state s , the following steps are performed.

1. The **closure** of state s is computed at Step 17. If a nonterminal B appears just after the bookmark symbol (\bullet), then in state s we can process a B once one has been found. Transitions from state s must include actions that can lead to discovery of a B. CLOSURE in Figure 6.9 returns a set that includes its supplied set of items along with fresh items for B's rules. The addition of fresh

```

function COMPUTELR0( Grammar ) : ( Set, State )
  States  $\leftarrow \emptyset$ 
  StartItems  $\leftarrow \{ \text{Start} \rightarrow \bullet \text{RHS}(p) \mid p \in \text{PRODUCTIONSFOR}(\text{Start}) \}$       7
  StartState  $\leftarrow \text{ADDSTATE}(\text{States}, \text{StartItems})$ 
  while ( s  $\leftarrow \text{WorkList.EXTRACTELEMENT}(\ ) \neq \perp$  ) do      8
    call COMPUTEGOTO( States, s )
    return ((States, StartState))
end
function ADDSTATE( States, items ) : State
  if items  $\notin$  States      9
  then
    s  $\leftarrow \text{newState}(\text{items})$       10
    States  $\leftarrow \text{States} \cup \{ s \}$ 
    WorkList  $\leftarrow \text{WorkList} \cup \{ s \}$       11
    Table[s][ $\star$ ]  $\leftarrow \text{error}$       12
  else s  $\leftarrow \text{FindState}(\text{items})$ 
  return (s)
end
function ADVANCEDOT( state,  $\mathcal{X}$  ) : Set
  return ( {  $A \rightarrow \alpha \mathcal{X} \bullet \beta \mid A \rightarrow \alpha \bullet \mathcal{X} \beta \in \text{state}$  } )      13
end

```

Figure 6.8: LR(0) construction.

items can trigger the addition of still more fresh items. Because the computed answer is a set, no item is added twice. The loop at Step 14 continues until nothing new is added. Thus, this loop eventually terminates.

- Step 18 determines transitions from s . When a new state is added during LR(0) construction, Step 12 sets all actions for this state as error. Transitions are defined for each grammar symbol \mathcal{X} that appears after the bookmark. COMPUTEGOTO in Figure 6.9 defines a transition at Step 20 from s to a (potentially new) state that reflects the parser's progress after shifting across every item in this state with \mathcal{X} after the bookmark. All such items indicate transition to the same state, because the parsers we construct must operate deterministically. In other words, the parse table has only one entry for a given state and symbol.

We now construct an LR(0) parse table for the grammar shown in Figure 6.1. In Figure 6.10 each state is shown as a separate box. The **kernel** of state s is the set of items explicitly represented in the state. We use the convention of drawing a line within a state to separate the kernel and closure items, as in States 0, 1, and 5; the other states did not require any closure items. Next to each item in each state is the state number reached by shifting the symbol next to the item's bookmark.

```

function CLOSURE(state) : Set
  ans ← state
  repeat
    prev ← ans
    foreach  $A \rightarrow \alpha \bullet B \gamma \in ans$  do
      foreach  $p \in \text{PRODUCTIONSFOR}(B)$  do
        ans ← ans  $\cup$  {  $B \rightarrow \bullet \text{RHS}(p)$  }
    until ans = prev
  return (ans)
end

procedure COMPUTEGOTO(States, s)
  closed ← CLOSURE(s)
  foreach  $\mathcal{X} \in (N \cup \Sigma)$  do
    RelevantItems ← ADVANCEDOT(closed,  $\mathcal{X}$ )
    if RelevantItems  $\neq \emptyset$ 
      then
        Table[s][ $\mathcal{X}$ ] ← shift ADDSTATE(States, RelevantItems)
  end

```

Figure 6.9: LR(0) closure and transitions.

In Figure 6.10 the transitions are also shown with labeled edges between the states. If a state contains a reducible item, then the state is double-boxed. The edges and double-boxed states emphasize that the basis for LR parsing is a **deterministic finite-state automaton** (DFA), called the **characteristic finite-state machine** (CFSM).

A **viable prefix** of a right sentential form is any prefix that does not extend beyond its handle. Formally, a CFSM recognizes its grammar's viable prefixes. Each transition shifts the symbols of a (valid) sentential form. When the automaton arrives in a double-boxed state, it has processed a viable prefix that ends with a handle. The handle is the RHS of the (unique) reducible item in the state. At this point, a reduction can be performed. The sentential form produced by the reduction can be processed anew by the CFSM. This process can be repeated until the grammar's goal symbol is shifted (successful parse) or the CFSM blocks (an input error).

For the input string plus plus num num num \$, Figure 6.11 shows the results of repeatedly presenting the (reverse) derived sentential forms to the CFSM. This approach serves to illustrate how a CFSM recognizes viable prefixes. However, it is wasteful to make repeated passes over an input string's sentential forms. For example, the repeated passes over the input string's first two tokens (plus plus) always cause the parser to enter State 1. Because the CFSM is deterministic, processing a given sequence of vocabulary symbols always has the same effect. Thus, the pars-

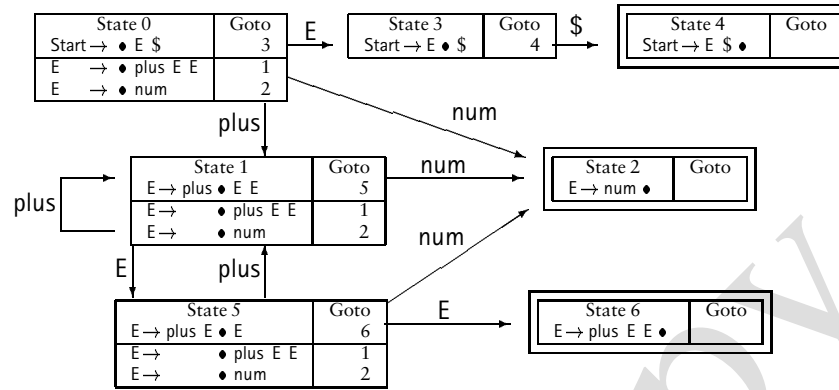


Figure 6.10: LR(0) computation for Figure 6.1

Sentential Prefix	Transitions	Resulting Sentential Form
		plus plus num num num \$
plus plus num	States 1, 1, and 2	plus plus E num num \$
plus plus E num	States 1, 1, 5, and 2	plus plus E E num \$
plus plus E E	States 1, 1, 5, and 6	plus E num \$
plus E num	States 1, 5, and 2	plus E E \$
plus E E	States 1, 5, and 6	E \$
E \$	States 1, 3, and 4	Start

Figure 6.11: Processing of plus plus num num num by the LR(0) machine in Figure 6.10.

ing algorithm given in Figure 6.3 does not make repeated passes over the derived sentential forms. Instead, the parse state is recorded after each shift so that current parse state is always associated with whatever symbol happens to be on top of stack. As reductions eat into the stack, the symbol exposed at the new top-of-stack bears the current state, which is the state the CFSM would reach if it rescanned the entire prefix of the current sentential form up to and including the current top-of-stack symbol.

If a grammar is LR(0), then the construction discussed in this section has the following properties:

- Given a syntactically correct input string, the CFSM will block in only double-boxed states, which call for a reduction. The CFSM clearly shows that no progress can occur unless a reduction takes place.
- There is at most one item present in any double-boxed state—the rule that should be applied upon entering the state. Upon reaching such states, the

```

procedure COMPLETETABLE( Table, grammar )
  call COMPUTELOOKAHEAD( )
  foreach state  $\in$  Table do
    foreach rule  $\in$  Productions(grammar) do
      call TRYRULEINSTATE( state, rule )
    call ASSERTENTRY( StartState, GoalSymbol, accept )           21
  end
procedure ASSERTENTRY( state, symbol, action )
  if Table[state][symbol] = error                                22
  then Table[state][symbol]  $\leftarrow$  action
  else
    call REPORTCONFLICT( Table[state][symbol], action )       23
  end

```

Figure 6.12: Completing an LR(0) parse table.

CFSM has completely processed a rule. The associated item is *reducible*, with the marker moved as far right as possible.

- If the CFSM's input string is syntactically invalid, then the parser will enter a state such that the offending terminal symbol cannot be shifted.

The table established during LR(0) construction at Step 20 in Figure 6.9 is almost suitable for parsing by the algorithm in Figure 6.3. Each state is a row of the table, and the columns represent grammar symbols. Entries are present only where the LR(0) construction allows transition between states—these are the shift actions. To complete the table, we apply the algorithm in Figure 6.12, which establishes the appropriate reduce actions.

For LR(0), the decision to call for a reduce is reflected in the code of Figure 6.13; arrival in a double-boxed state signals a reduction irrespective of the next input token. As reduce actions are inserted, ASSERTENTRY reports any *conflicts* that arise when a given state and grammar symbol call for multiple parsing actions. Step 22 allows an action to be asserted only if the relevant table cell was previously undefined (*error*). Finally, Step 21 calls for acceptance when the goal symbol is shifted in the table's start state. Given the construction in Figure 6.10 and the grammar in Figure 6.1, LR(0) analysis yields the parse table is shown in Figure 6.14.

6.4 Conflict Diagnosis

Sometimes LR construction is not successful, even for simple languages and grammars. In the following sections we consider table-construction methods that are

```

procedure COMPUTELOOKAHEAD( )
  /* Reserved for the LALR(k) computation given in Section 6.5.2 */
end
procedure TRYRULEINSTATE(s, r)
  if LHS(r) → RHS(r) • ∈ s
  then
    foreach  $\mathcal{X} \in (\Sigma \cup N)$  do call ASSERTENTRY(s,  $\mathcal{X}$ , reduce r)
  end

```

Figure 6.13: LR(0) version of TRYRULEINSTATE.

State	num	plus	\$	Start	E
0	2	1		accept	3
1	2	1			5
2	reduce 3				
3			4		
4	reduce 1				
5	2	1			6
6	reduce 2				

Figure 6.14: LR(0) parse table for the grammar in Figure 6.1.

more powerful than LR(0), thereby accommodating a much larger class of grammars. In this section, we examine why *conflicts* arise during LR table construction. We develop approaches for understanding and resolving such conflicts.

The generic LR parser shown in Figure 6.3 is *deterministic*. Given a parse state and an input symbol, the parse table can specify exactly one action to be performed by the parser—shift, reduce, accept, or error. In Chapter Chapter:global:three we tolerated nondeterminism in the scanner specification because we knew of an efficient for transforming a nondeterministic DFA into a deterministic DFA. Unfortunately, no such algorithm is possible for stack-based parsing engines. Some CFGs cannot be parsed deterministically. In such cases, *perhaps* there is another grammar that generates the same language, but for which a deterministic parser can be constructed. There are **context-free languages** (CFLs) that provably cannot be parsed using the (deterministic) LR method (see Exercise 9). However, programming languages are typically designed to be parsed deterministically.

A parse-table **conflict** arises when the table-construction method cannot decide between multiple alternatives for some table-cell entry. We then say that the associated state (row of the parse table) is **inadequate** for that method. An inadequate state for a weaker table-construction algorithm can sometimes be resolved by a

stronger algorithm. For example, the grammar of Figure 6.4 is not LR(0)—a mix of shift and reduce actions can be seen in State 0. However, the table-construction algorithms introduced in Section 6.5 resolve the LR(0) conflicts for this grammar.

If we consider the possibilities for multiple table-cell entries, only the following two cases are troublesome for LR(k) parsing.

- **shift/reduce conflicts** exist in a state when table construction cannot use the next k tokens to decide whether to shift the next input token or call for a reduction. The bookmark symbol must occur before a terminal symbol t in one of the state's items, so that a shift of t could be appropriate. The bookmark symbol must also occur at the end of some other item, so that a reduction in this state is also possible.
- **reduce/reduce conflicts** exist when table construction cannot use the next k tokens to distinguish between multiple reductions that could be applied in the inadequate state. Of course, a state with such a conflict must have at least two reducible items.

Other combinations of actions in a table cell do not make sense. For example, it cannot be the case that some terminal t could be shifted but also cause an error. Also, we cannot obtain a shift/shift error: if a state admits the shifting of terminal symbols t and u , then the target state for the two shifts is different, and there is no conflict. Exercise 11 considers the impossibility of a shift/reduce conflict on a nonterminal symbol.

Although the table-construction methods we discuss in the following sections vary in power, each is capable of reporting conflicts that render a state inadequate. Conflicts arise for one of the following reasons.

- The grammar is *ambiguous*. No (deterministic) table-construction method can resolve conflicts that arise due to ambiguity. Ambiguous grammars are considered in Section 6.4.1, but here we summarize possibilities for addressing the ambiguity. If a grammar is ambiguous, then some input string has at least two distinct parse trees.
 - If both parse trees are desirable (as in Exercise 37), then the grammar's language contains a *pun*. While puns may be tolerable in natural languages, they are undesirable in the design of computer languages. A program specified in a computer language should have an unambiguous interpretation.
 - If only one tree has merit, then the grammar can often be modified to eliminate the ambiguity. While there are inherently ambiguous languages (see Exercise 12), computer languages are not designed with this property.
- The grammar is not ambiguous, but the current table-building approach could not resolve the conflict. In this case, the conflict might disappear if one or more of the following approaches is followed:

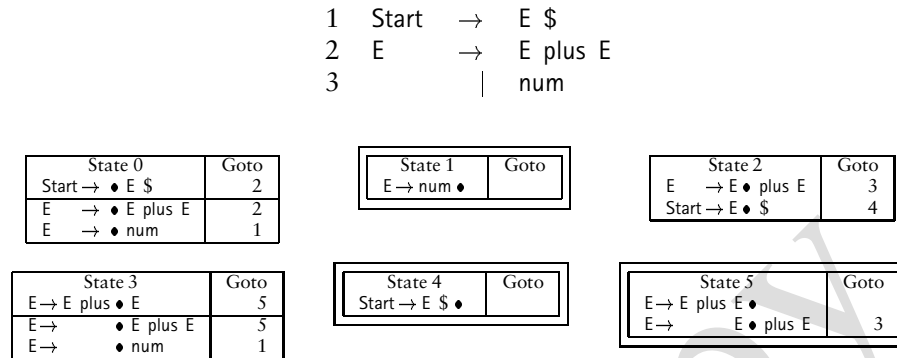


Figure 6.15: An ambiguous expression grammar.

- The current table-construction method is given more lookahead.
- A more powerful table-construction method is used.

It is possible that no amount of lookahead or table-building power can resolve the conflict, even if the grammar is unambiguous. We consider such a grammar in Section 6.4.2 and in Exercise 36.

When an $LR(k)$ construction algorithm develops an inadequate state, it is an unfortunate but important fact is that it is not possible automatically to decide which of the above problem afflicts the grammar [?]. It is impossible to construct an algorithm to determine if a CFG is ambiguous. It is therefore also impossible to determine whether a bounded amount of lookahead can resolve an inadequate state. As a result, human (instead of mechanical) reasoning is required to understand and repair grammars for which conflicts arise. Sections 6.4.1 and 6.4.2 develop intuition and strategies for such reasoning.

6.4.1 Ambiguous Grammars

Consider the grammar and its $LR(0)$ construction shown in Figure 6.15. The grammar generates sums of numbers using the familiar *infix* notation. In the $LR(0)$ construction, all states are adequate except State 5. In this state a plus can be shifted to arrive in State 3. However, State 5 also allows reduction by the rule $E \rightarrow E \text{ plus } E$. This inadequate state exhibits a shift/reduce conflict for $LR(0)$. To resolve this conflict, it must be decided how to fill in the LR parse table for State 5 and the symbol plus. Unfortunately, this grammar is ambiguous, so a unique entry cannot be determined.

While there is no automatic method for determining if an arbitrary grammar is ambiguous, the inadequate states can provide valuable assistance in finding a string with multiple derivations—should one exist. Recall that a parser state represents transitions made by the CFM when recognizing viable prefixes. The bookmark

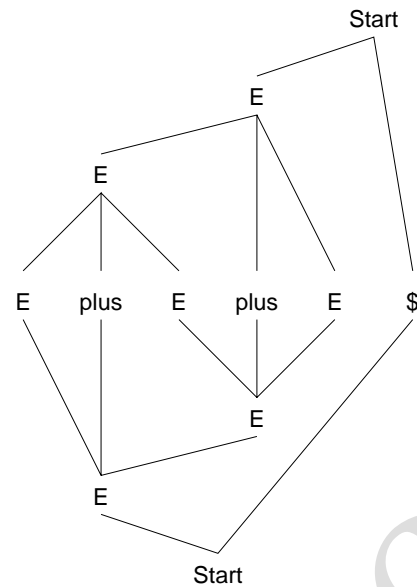


Figure 6.16: Two derivations for $E \text{ plus } E \text{ plus } E \text{ \$}$. The parse tree on top favors reduction in State 5; the parse tree on bottom favors a shift.

symbol shows the progress made so far; symbols appearing after the bookmark are symbols that can be shifted to make progress toward a successful parse. While our ultimate goal is the discovery of an input string with multiple derivations, we begin by trying to find an ambiguous sentential form. Once identified, the sentential form can easily be extended into a terminal string, by replacing nonterminals using the grammar's productions.

Using State 5 in Figure 6.15 as an example, the steps taken to understand conflicts are as follows.

1. Using the parse table or CFSM, determine a sequence of vocabulary symbols that cause the parser to move from the start state to the inadequate state. For Figure 6.15, the simplest such sequence is $E \text{ plus } E$, which passes through States 0, 2, 3, and 5. Thus, in State 5 we have $E \text{ plus } E$ on the top-of-stack; one option is a reduction by $E \rightarrow E \text{ plus } E$. However, with the item $E \rightarrow E \bullet \text{ plus } E$ it is also possible to shift a plus and then an E .
2. If we line up the dots of these two items, we obtain a snapshot of what is on the stack upon arrival in this state and what may be successfully shifted in the future. Here we obtain the sentential form prefix $E \text{ plus } E \bullet \text{ plus } E$. The shift/reduce conflict tells us that there are two potentially successful parses. We therefore try to construct two derivation trees for $E \text{ plus } E \text{ plus } E$, one assuming the reduction at the bookmark symbol and one assuming the shift.

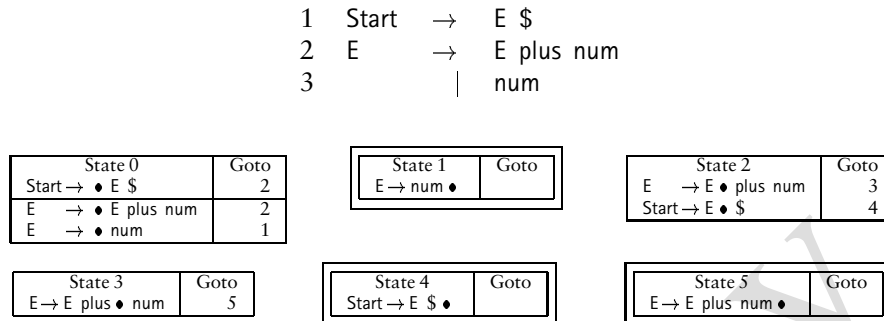


Figure 6.17: Unambiguous grammar for infix sums and its LR(0) construction.

Completing either derivation may require extending this sentential prefix so that it becomes a sentential form: a string of vocabulary symbols derivable (in two different ways) from the goal symbol.

For our example, E plus E plus E is almost a complete sentential form. We need only append \$ to obtain E plus E plus E \$.

To emphasize that the derivations are constructed for the same string, Figure 6.16 shows the derivations above and below the sentential form. If the reduction is performed, then the early portion of E plus E plus E \$ is structured under a nonterminal; otherwise, the input string is shifted so that the latter portion of the sentential form is reduced first. The parse tree that favors the reduction in State 5 corresponds to **left-association** for addition, while the shift corresponds to **right-association**.

Having analyzed the ambiguity in the grammar of Figure 6.15, we next eliminate the ambiguity by creating a grammar that favors left-association—the reduction instead of the shift. Such a grammar and its LR(0) construction are shown in Figure 6.17. The grammars in Figures 6.15 and 6.17 generate the same language. In fact, the language is regular, denoted by the regular expression $\text{num (plus num)}^* \$$. So we see that even simple languages can have ambiguous grammars. In practice, diagnosing ambiguity can be more difficult. In particular, finding the ambiguous sentential form may require significant extension of a viable prefix. Exercises 36 and 37 provide practice in finding and fixing a grammar's ambiguity.

6.4.2 Grammars that are not LR(k)

Figure 6.18 shows a grammar and a portion of its LR(0) construction for a language similar to infix addition, where expressions end in either a or b. The complete LR(0) construction is left as Exercise 13. State 2 contains a reduce/reduce conflict. In this state, it is not clear whether num should be reduced to an E or an F. The viable prefix that takes us to State 2 is simply num. To obtain a sentential form, this must be extended either to num a \$ or num b \$. If we use the former sentential form,

1	Start	→	Exprs \$
2	Exprs	→	E a
3			F b
4	E	→	E plus num
5			num
6	F	→	F plus num
7			num

State 0		Goto
Start → • Exprs \$		1
Exprs → • E a		4
Exprs → • F b		3
E → • E plus num		4
E → • num		2
F → • F plus num		3
F → • num		2

State 2		Goto
E → num •		
F → num •		

Figure 6.18: A grammar that is not LR(k).

then F cannot be involved in the derivation. Similarly, if we use the latter sentential form, E is not involved. Thus, progress past num cannot involve more than one derivation, and the grammar is not ambiguous.

Since LR(0) construction failed for the grammar in Figure 6.18, we could try a more ambitious table-construction method from among those discussed in Sections 6.5 and 6.6. It turns out that none can succeed. All LR(k) constructions analyze grammars using k lookahead symbols. If a grammar is LR(k), then there is some value of k for which all states are adequate in the LR(k) construction described in Section 6.6. The grammar in Figure 6.18 is not LR(k) for any k . To see this, consider the following rightmost derivation of num plus ... plus num a.

Start	⇒ _{rm}	Exprs \$
	⇒ _{rm}	E a \$
	⇒ _{rm}	E plus num a \$
	⇒ _{rm} [*]	E plus ... plus num a \$
	⇒ _{rm}	num plus ... plus num a \$

A bottom-up parse must play the above derivation backwards. Thus, the first few steps of the parse will be:

0	Initial Configuration	num plus plus num a \$
0	num 2	shift num plus plus num a \$

With num on top-of-stack, we are in State 2. A deterministic, bottom-up parser must decide at this point whether to reduce num to an E or an F. If the decision were delayed, then the reduction would have to take place in the middle of the

1	Start	→	E \$
2	E	→	E plus num
3			E times num
4			num

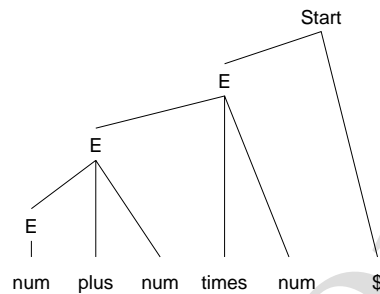


Figure 6.19: Expressions with sums and products.

stack, and this is not allowed. The information needed to resolve the reduce/reduce conflict appears just before the \$ symbol. Unfortunately, the relevant a or b could be arbitrarily far ahead in the input, because strings derived from E or F can be arbitrarily long.

In summary, simple grammars and languages can have subtle problems that prohibit generation of a bottom-up parser. It follows that a top-down parser would also fail on such grammars (Exercise 14). The LR(0) construction can provide important clues for diagnosing a grammar's inadequacies; understanding and resolving such conflicts requires human intelligence. Also, the LR(0) construction forms the basis of the SLR(*k*) and LALR(*k*) constructions—techniques we consider next.

6.5 Conflict Resolution for LR(0) Tables

While LR(0) construction succeeded for the grammar in Figure 6.17, most grammars require some lookahead during table construction. In this section we consider methods that are based on the LR(0) construction. Where conflicts arise, these methods analyze the grammar to determine if the conflict can be resolved without expanding the number of states (rows) in the LR(0) table. Section 6.5.1 presents the SLR(*k*) construction, which is simple but does not work as well as the LALR(*k*) construction introduced in Section 6.5.2.

6.5.1 SLR(*k*) Table Construction

The SLR(*k*) method attempts to resolve inadequate states using grammar analysis methods presented in Chapter Chapter:global:four. To demonstrate the SLR(*k*) construction, we require a grammar that is not LR(0). We begin by extending the

grammar in Figure 6.17 to accommodate expressions involving sums and products. Figure 6.19 shows such a grammar along with a parse tree for the string num plus num times num \$. Exercise 15 shows that this grammar is LR(0). However, it does not produce the parse trees that structure the expressions appropriately. The parse tree shown in Figure 6.19 structures the computation by adding the first two nums and then multiplying that sum by the third num. As such, the input string $3+4*7$ would produce a value of 49 if evaluation were guided by the computation's parse tree.

A common convention in mathematics is that multiplication has precedence over addition. Thus, the computation $3 + 4 * 7$ should be viewed as adding 3 to the product $4 * 7$, resulting in the value 31. Such conventions are typically adopted in programming language design, in an effort to simplify program authoring and readability. We therefore seek a parse tree that appropriately structures expressions involving multiplication and addition.

To develop the grammar that achieves the desired effect, we first observe that a string in the language of Figure 6.19 should be regarded as a *sum of products*. The grammar in Figure 6.17 generates sums of nums. A common technique to expand a language involves replacing a terminal symbol in the grammar by a nonterminal whose role in the grammar is equivalent. To produce a sum of Ts rather than a sum of nums, we need only replace num with T to obtain the rules for E shown in Figure 6.20. To achieve a sum of products, each T can now derive a product, with the simplest product consisting of a single num. Thus, the rules for T are based on the rules for E, substituting times for plus. Figure 6.20 shows a parse tree for the input string from Figure 6.19, with multiplication having precedence over addition.

Figure 6.21 shows a portion of the LR(0) construction for our precedence-respecting grammar. States 1 and 6 are inadequate for LR(0): in each of these states, there is the possibility of shifting a times or applying a reduction to E. Figure 6.21 shows a sequence of parser actions for the sentential form E plus num times num \$, leaving the parser in State 6.

Consider the shift/reduce conflict of State 6. To determine if the grammar in Figure 6.20 is ambiguous, we turn to the methods described in Section 6.4. We proceed by assuming the shift and reduce are *each* possible given the sentential form E plus T times num \$.

- If the shift is taken, then we can continue the parse in Figure 6.21 to obtain the parse tree shown in Figure 6.20.
- Reduction by rule $E \rightarrow E \text{ plus } T$ yields E times num \$, which causes the CFSM in Figure 6.21 to block in State 3 with no progress possible. If we try to reduce using $T \rightarrow \text{num}$, then we obtain E times T \$, which can be further reduced to E times E \$. Neither of phrases can be further reduced to the goal symbol.

Thus, E times num \$ is not a valid sentential form for this grammar and a reduction in State 6 for this sentential form is inappropriate.

With the item $E \rightarrow E \text{ plus } T \bullet$ in State 6, reduction by $E \rightarrow E \text{ plus } T$ must be appropriate under some conditions. If we examine the sentential forms E plus T \$ and

1	Start	→	E \$
2	E	→	E plus T
3			T
4	T	→	T times num
5			num

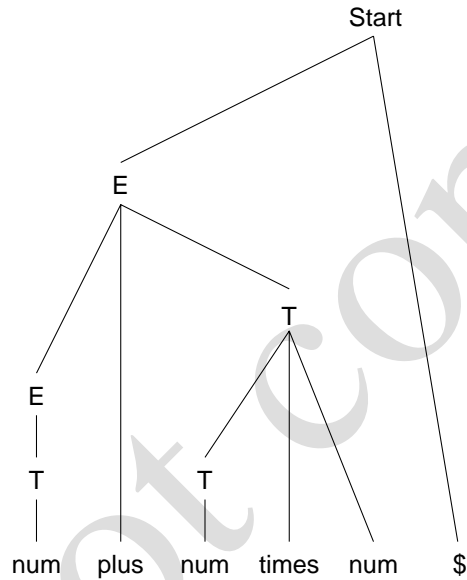


Figure 6.20: Grammar for sums of products.

$E \text{ plus } T \text{ plus num } \$$, we see that the $E \rightarrow E \text{ plus } T$ must be applied in State 6 when the next input symbol is plus or \$, but not times. LR(0) could not selectively call for a reduction in any state; perhaps methods that can consult lookahead information in TRYRULEINSTATE can resolve this conflict.

Consider the sequence of parser actions that could be applied between a reduction by $E \rightarrow E \text{ plus } T$ and the next shift of a terminal symbol. Following the reduction, E must be shifted onto the stack. At this point, assume terminal symbol plus is the next input symbol. If the reduction to E can lead to a successful parse, then plus can appear next to E in *some* valid sentential form. An equivalent statement is $\text{plus} \in \text{Follow}(E)$, using the Follow computation from Chapter Chapter:global:four.

SLR(k) parsing uses $\text{Follow}_k(A)$ to call for a reduction to A in any state containing a reducible item for A . Algorithmically, we obtain SLR(k) by performing the LR(0) construction in Figure 6.8; the only change is to the method TRYRULEINSTATE, whose SLR(1) version is shown in Figure 6.22. For our example, States 1 and 6 are resolved by computing $\text{Follow}(E) = \{\text{plus}, \$\}$. The SLR(1) parse table that results from this analysis is shown in Figure 6.23.

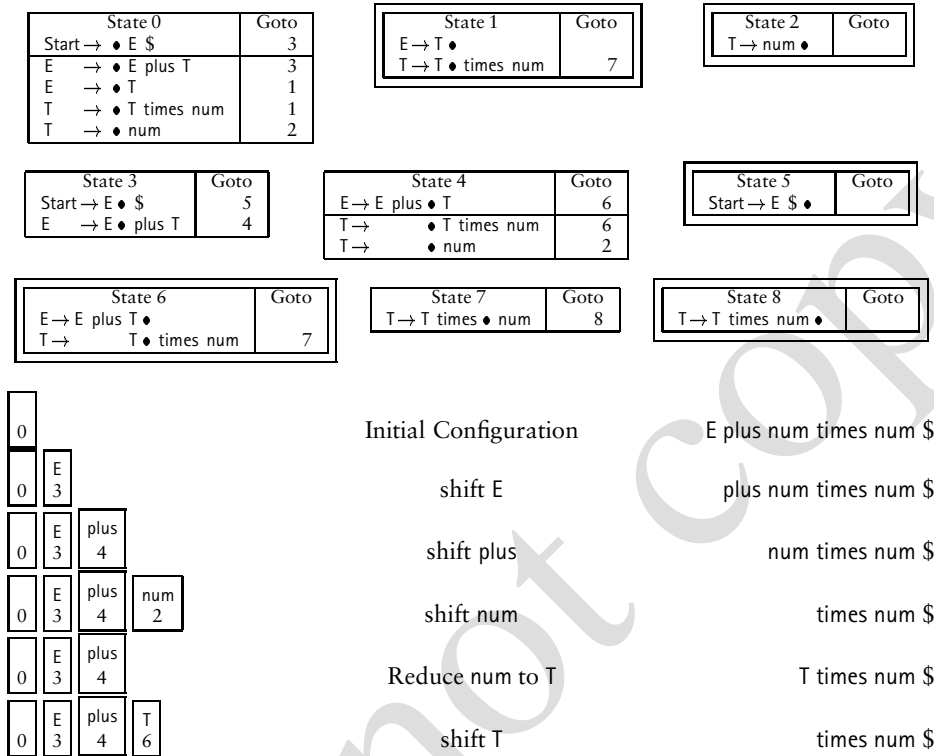


Figure 6.21: LR(0) construction and parse leading to inadequate State 6.

```

procedure TRYRULEINSTATE(s, r)
  if LHS(r)  $\rightarrow$  RHS(r)  $\bullet \in s$ 
  then
    foreach  $\mathcal{X} \in \text{Follow}(\text{LHS}(r))$  do
      call ASSERTENTRY(s,  $\mathcal{X}$ , reduce r)
  end
  
```

Figure 6.22: SLR(1) version of TRYRULEINSTATE.

State	num	plus	times	\$	Start	E	T
0	2				accept	3	1
1		3	7	3			
2		5	5	5			
3		4		5			
4	2						6
5				1			
6		2	7	2			
7	8						
8		4	4	4			

Figure 6.23: SLR(1) parse table for the grammar in Figure 6.20.

6.5.2 LALR(k) Table Construction

SLR(k) attempts to resolve LR(0) inadequate states using Follow_k information. Such information is computed *globally* for a grammar. Sometimes SLR(k) construction fails only because the Follow_k information is not *rule-specific*. Consider the grammar and its partial LR(0) construction shown in Figure 6.24. This grammar generates the language $\{a, ab, ac, xac\}$. The grammar is not ambiguous, because each of these strings has a unique derivation. However, State 3 has an LR(0) shift/reduce conflict. SLR(k) tries to resolve the shift/reduce conflict, computing $\text{Follow}_k(A) = \{b\$^{k-1}, c\$^{k-1}, \$^k\}$. A can be followed in some sentential form by any number of end-of-string symbols, possibly prefaced by b or c . If we examine States 0 and 3 more carefully, we see that it is not possible for c to occur after the expansion of A in State 3. In the closure of State 0, the fresh item for A was created by the item $S \rightarrow \bullet A B$. Following the shift of A , only b or $\$$ can occur—there is no sentential form $A c \$$. Given the above analysis, we can fix the shift/reduce conflict by modifying the grammar to have two “versions” of A . Using A_1 and A_2 , the resulting SLR(1) grammar is shown in Figure 6.25. Here, State 2 is resolved, since $\text{Follow}(A_1) = \{\$, b\}$.

SLR(k) has difficulty with the grammar in Figure 6.24 because Follow sets are computed *globally* across the grammar. Copying productions and renaming non-terminals can cause the Follow computation to become more production-specific, as in Figure 6.25. However, this is tedious and can complicate the understanding and maintenance of a programming language's grammar. In this section we consider LALR(k) parsing, which offers a more specialized computation of “follow” information. Short for “lookahead” LR, the term LALR is not particularly informative—SLR and LR also use lookahead. However, LALR offers superior lookahead analysis

1 Start \rightarrow S \$
 2 S \rightarrow A B
 3 | a c
 4 | x A c
 5 A \rightarrow a
 6 B \rightarrow b
 7 | λ

State 0	Goto
Start \rightarrow • S \$	4
S \rightarrow • A B	2
S \rightarrow • a c	3
S \rightarrow • x A c	1
A \rightarrow • a	3

State 3	Goto
S \rightarrow a • c	6
A \rightarrow a •	

Figure 6.24: A grammar that is not SLR(k).

1 Start \rightarrow S \$
 2 S \rightarrow A₁ B
 3 | a c
 4 | x A₂ c
 5 A₁ \rightarrow a
 6 A₂ \rightarrow a
 7 B \rightarrow b
 8 | λ

State 0	Goto
Start \rightarrow • S \$	3
S \rightarrow • A ₁ B	4
S \rightarrow • a c	2
S \rightarrow • x A ₂ c	1
A ₁ \rightarrow • a	2

State 2	Goto
S \rightarrow a • c	8
A ₁ \rightarrow a •	

Figure 6.25: An SLR(1) grammar for the language defined in Figure 6.24.

for the LR(0) table.

Like $SLR(k)$, $LALR(k)$ is based on the LR(0) construction given in Section 6.3. Thus, an $LALR(k)$ table has the same number of rows (states) as does an LR(0) table for the same grammar. While LR(k) (discussed in Section 6.6) offers more powerful lookahead analysis, this is achieved at the expense of introducing more states.

Due to its balance of power and efficiency, $LALR(1)$ is the most popular LR table-building method. Chapter Chapter:global:seven describes tools that can automatically construct parsers for grammars that are $LALR(1)$. For $LALR(1)$, we redefine the following two methods from Figure 6.13.

COMPUTELOOKAHEAD Figure 6.26 contains code to build and evaluate a *lookahead propagation graph*. This computation establishes $ItemFollow((state, item))$ as the set of (terminal) symbols that can follow the reducible *item* as of the given *state*. The propagation graph is described below in greater detail.

TRYRULEINSTATE In the $LALR(1)$ version, the *ItemFollow* set for a reducible item is consulted for symbols that can apparently occur after this reduction.

Propagation Graph

We have not formally named the LR(0) items, but an item occurs at most once in any state. Thus, the pair $(s, A \rightarrow \alpha \bullet \beta)$ suffices to identify the item $A \rightarrow \alpha \bullet \beta$ that occurs in state s . For each valid state and item pair, Step 24 in Figure 6.26 creates a vertex v in the propagation graph. Each item in an LR(0) construction is represented by a vertex in this graph. The *ItemFollow* sets are initially empty, except for the augmenting item $Start \rightarrow \bullet S \$$ in the LR(0) start-state. Edges are placed in the graph between items i and j when the symbols that follow the reducible form of item i should be included in the corresponding set of symbols for item j . For the purposes of lookahead analysis, the input string can be considered to have an arbitrary number of “end-of-string” characters. Step 25 forces the entire program, derived from any rule for Start, to be followed by \$.

Step 26 of the algorithm in Figure 6.26 considers items of the form $A \rightarrow \alpha \bullet B\gamma$ in state s . This generic item indicates that the bookmark is just before the symbol B , with grammar symbols in α appearing before the bookmark, and grammar symbols in γ appearing after B . Note that α or γ could be absent, in which case $\alpha = \lambda$ or $\gamma = \lambda$, respectively. The symbol B is always present, unless the grammar rule is $A \rightarrow \lambda$. The lookahead computation is specifically concerned with $A \rightarrow \alpha \bullet B\gamma$ when B is a nonterminal, because CLOSURE in Figure 6.9 adds items to state s for each of B 's productions. The symbols that can follow B depend on γ , which is either absent or present in the item. Also, even when γ is present, it is possible that $\gamma \Rightarrow^* \lambda$. The algorithm in Figure 6.26 takes these cases into account as follows.

- For the item $A \rightarrow \alpha \bullet B\gamma$, any symbol in $First(\gamma)$ can follow each closure item $B \rightarrow \bullet \delta$. This is true even when γ is absent: in such cases, $First(\lambda) = \emptyset$. Thus, Step 28 places symbols from $First(\gamma)$ in the *ItemFollow* sets for each $B \rightarrow \bullet \delta$ in state s .

```

procedure COMPUTELOOKAHEAD( )
  call BUILDITEMPROPGRAPH( )
  call EVALITEMPROPGRAPH( )
end
procedure BUILDITEMPROPGRAPH( )
  foreach  $s \in States$  do
    foreach  $item \in state$  do
       $v \leftarrow Graph.ADDVERTEX((s, item))$  24
       $ItemFollow(v) \leftarrow \emptyset$ 
    foreach  $p \in PRODUCTIONSFOR(Start)$  do
       $ItemFollow((StartState, Start \rightarrow \bullet RHS(p))) \leftarrow \{ \$ \}$  25
    foreach  $s \in States$  do
      foreach  $A \rightarrow \alpha \bullet B \gamma \in s$  do 26
         $v \leftarrow Graph.FINDVERTEX((s, A \rightarrow \alpha \bullet B \gamma))$ 
        call  $Graph.ADDEDGE(v, (Table[s][B], A \rightarrow \alpha B \bullet \gamma))$  27
        foreach  $(w \leftarrow (s, B \rightarrow \bullet \delta)) \in Graph.Vertices$  do
           $ItemFollow(w) \leftarrow ItemFollow(w) \cup First(\gamma)$  28
          if ALLDERIVEEMPTY( $\gamma$ ) 29
            then call  $Graph.ADDEDGE(v, w)$ 
        end
      end
    end
  end
procedure EVALITEMPROPGRAPH( ) 30
  repeat
     $changed \leftarrow false$ 
    foreach  $(v, w) \in Graph.Edges$  do
       $old \leftarrow ItemFollow(w)$ 
       $ItemFollow(w) \leftarrow ItemFollow(w) \cup ItemFollow(v)$ 
      if  $ItemFollow(w) \neq old$ 
        then  $changed \leftarrow true$ 
    until not  $changed$ 
  end

```

Figure 6.26: LALR(1) version of COMPUTELOOKAHEAD.

```

procedure TRYRULEINSTATE( $s, r$ )
  if  $LHS(r) \rightarrow RHS(r) \bullet \in s$ 
    then
      foreach  $\mathcal{X} \in \Sigma$  do
        if  $\mathcal{X} \in ItemFollow((s, LHS(r) \rightarrow RHS(r) \bullet))$ 
          then call ASSERTENTRY( $s, \mathcal{X}, reduce\ r$ )
      end
  end

```

Figure 6.27: LALR(1) version of TRYRULEINSTATE.

State	LR(0) Item	Goto State	Prop Edges Placed by Step		Initialize <i>ItemFollow</i>	
			27	29	First(γ)	28
0	1 Start $\rightarrow \bullet$ S \$	4	13		\$	2,3,4
	2 S $\rightarrow \bullet$ A B	2	8	5	b	5
	3 S $\rightarrow \bullet$ a c	3	11			
	4 S $\rightarrow \bullet$ x A c	1	6			
	5 A $\rightarrow \bullet$ a	3	12			
1	6 S \rightarrow x \bullet A c	9	18		c	7
	7 A $\rightarrow \bullet$ a	10	19			
2	8 S \rightarrow A \bullet B	8	17	9,10		
	9 B $\rightarrow \bullet$ b	7	16			
	10 B $\rightarrow \bullet$					
3	11 S \rightarrow a \bullet c	6	15			
	12 A \rightarrow a \bullet					
4	13 Start \rightarrow S \bullet \$	5	14			
5	14 Start \rightarrow S \$ \bullet					
6	15 S \rightarrow a c \bullet					
7	16 B \rightarrow b \bullet					
8	17 S \rightarrow A B \bullet					
9	18 S \rightarrow x A \bullet c	11	20			
10	19 A \rightarrow a \bullet					
11	20 S \rightarrow x A c \bullet					

Figure 6.28: LALR(1) analysis of the grammar in Figure 6.24.

ItemFollow sets are useful only when the bookmark progresses to the point of a reduction. $B \rightarrow \bullet \delta$ is only the *promise* of a reduction to B once δ has been found. Thus, the lookahead symbols must accompany the bookmark's progress across δ so they are available for $B \rightarrow \delta \bullet$ in the appropriate state. Such migration of lookahead symbols is represented by propagation edges, as described next.

- Edges must be placed in the propagation graph when the symbols that are associated with one item should be added to the symbols associated with another item. Following are two situations that call for the addition of propagation edges.
 - As described above, the lookahead symbols introduced by Step 28 are useful only when the bookmark has advanced to the end of the rule. In LR(0), the CFSM contains an edge from state s to state t when the ad-

vance of the bookmark symbol for an item in state s creates an item in state t . For lookahead propagation in LALR, the edges are more specific—Step 27 places edges in the propagation graph between *items*. Specifically, an edge is placed from an item $A \rightarrow \alpha \bullet B \gamma$ in state s to the item $A \rightarrow \alpha B \bullet \gamma$ in state t obtained by advancing the bookmark, if t is the CFSM state reached by processing a B in state s .

- Consider again the item $A \rightarrow \alpha \bullet B \gamma$ and the closure items introduced when B is a nonterminal. When $\gamma \Rightarrow^* \lambda$, either because γ is absent or because the string of symbols in γ can derive λ , then any symbol that can follow A can also follow B . Thus, Step 29 places a propagation edge from the item for A to the item for B .

The edges placed by these steps are used at Step 30 to affect the appropriate *ItemFollow* sets. The loop at Step 30 continues until no changes are observed in any *ItemFollow* set. This loop must eventually terminate, because the lookahead sets are only increased, by symbols drawn from a finite alphabet (Σ).

Now consider the grammar in Figure 6.24 and its LALR(1) construction shown in Figure 6.28. The items listed under the column for Step 27 are the targets of edges placed in the propagation graph to carry symbols to the point of reduction. For example, consider Items 6 and 7. For the item $S \rightarrow x \bullet A$ c , we have $\gamma = c$. Thus, when the item $A \rightarrow \bullet a$ is generated in Item 7, c can follow the reduction to A . Step 28 therefore adds c directly to Item 5's *ItemFollow* set. However, c is not useful until it is time to apply the reduction $A \rightarrow a$. Thus, propagation edges are placed between Items 7 and 19 by Step 27.

In most cases, lookahead is either *generated* (when $\text{First}(\gamma) \neq \emptyset$) or *propagated* (when $\gamma = \lambda$). However, it is possible that $\text{First}(\gamma) \neq \emptyset$ and $\gamma \Rightarrow^* \lambda$, as in Item 2. Here, $\gamma = B$; we have $\text{First}(B) = \{b\}$ but we also have $B \Rightarrow^* \lambda$. Thus, Step 28 causes b to contribute to Item 5's *ItemFollow* set. Additionally, the *ItemFollow* set at Item 2 is forwarded to Item 5 by a propagation edge placed by Step 29. Finally, the lookahead present at Item 2 must make its way to Item 17 where it can be consulted when the reduction $S \rightarrow AB$ is applied. Thus, propagation edges are placed by Step 27 between Items 2 and 8 and between Items 8 and 17.

Constructing the propagation graph is only half of the process we need to compute lookahead sets. Once LALR(1) construction has established the propagation graph and has initialized the *ItemFollow* sets, as shown in Figure 6.28, the propagation graph can be evaluated. EVALITEMPROPGRAPH in Figure 6.26 evaluates the graph by iteratively propagating lookahead information along the graph's edges until no new information appears.

In Figure 6.29 we trace the progress of this algorithm on our example. The “Initial” column shows the lookahead sets established by Step 28. The loop at Step 30 unions lookahead sets as determined by the propagation graph's edges. For the example we have considered thus far, the loop at Step 30 converges after a single pass. As written, the algorithm requires a second pass to detect that no

Item	Prop To	Initial	Pass 1
1	13	\$	
2	5,8	\$	
3	11	\$	
4	6	\$	
5	12	b	\$
6	18		\$
7	19	c	
8	9,10,17		\$
9	16		\$
10			\$
11	15		\$
12			b \$
13	14		\$
14			\$
15			\$
16			\$
17			\$
18	20		\$
19			c
20			\$

Figure 6.29: Iterations for LALR(1) follow sets.

lookahead sets change after the first pass. We do not show the second pass in Figure 6.29.

The loop at Step 30 continues until no *ItemFollow* set is changed from the previous iteration. The number of iterations prior to convergence depends on the structure of the propagation graph. The graph with edges specified in Figure 6.29 is *acyclic*—such graphs can be evaluated completely in a single pass.

In general, multiple passes can be required for convergence. We illustrate this using Figure 6.31, which records how an LALR(1) propagation graph is constructed for the grammar shown in Figure 6.30. Figure 6.32 shows the progress of the loop at Step 30. The lookahead sets for a given item are the union of the symbols displayed in the three right-most columns. For this example, the sets converge after two passes, but a third pass is necessary to detect this. Two passes are necessary, because the propagation graph embeds the graph shown in Figure 6.33. This graph contains a cycle with one “retreating” backedge. Information cannot propagate from item 20 to 12 in a single pass. Exercise 27 explores how to extend the grammar in Figure 6.30 so that convergence can require *any number* of iterations. In practice, LALR(1) lookahead computations converge quickly, usually in one or two passes.

In summary, LALR(1) is a powerful parsing method and is the basis for most

1	Start	→	S \$
2	S	→	x C1 y1 Cn yn
3			A1
4	A1	→	b1 C1
5			a1
6	An	→	bn Cn
7			an
8	C1	→	An
9	Cn	→	A1

Figure 6.30: LALR(1) analysis: grammar.

bottom-up parser generators. To achieve greater power, more lookahead can be applied, but this is rarely necessary. LALR(1) grammars are available for all popular programming languages.

6.6 LR(k) Table Construction

In this section we describe an LR table-construction method that accommodates all deterministic, context-free languages. While this may seem a boon, LR(k) parsing is not very practical, because even LR(1) tables are typically much larger than the LR(0) tables upon which SLR(k) and LALR(k) parsing are based. Moreover, it is rare that LR(1) can handle a grammar for which LALR(1) construction fails. We present such a grammar in Figure 6.34, but grammars such as these do not arise very often in practice. When LALR(1) fails, it is typically for one of the following reasons.

- The grammar is ambiguous—LR(k) cannot help.
- More lookahead is needed—LR(k) can help (for $k > 1$). However, LALR(k) would probably also work in such cases.
- No amount of lookahead suffices—LR(k) cannot help.

The grammar in Figure 6.34 allows strings generated by the nonterminal M to be surrounded by *matching* parentheses (lp and rp) or braces (lb and rb). The grammar also allows S to generate strings with *unmatched* punctuation. The unmatched expressions are generated by the nonterminal U . The grammar can easily be expanded by replacing the terminal $expr$ with a nonterminal that derives arithmetic expressions, such as E in the grammar of Figure 6.20. While M and U generate the same terminal strings, the grammar distinguishes between them so that a semantic action can report the mismatched punctuation—using reduction by $U \rightarrow expr$.

A portion of the LALR(1) analysis of the grammar in Figure 6.34 is shown in Figure 6.36; the complete analysis is left for Exercise 28. Consider the lookaheads that will propagate into State 6. For Item 14, which calls for the reduction

State	LR(0) Item	Goto State	Prop Edges Placed by Step		Initialize <i>ItemFollow</i>	
			27	29	First(γ)	28
0	1 Start $\rightarrow \bullet$ S \$	3	11		\$	2,3
	2 S $\rightarrow \bullet$ x C1 y1 Cn yn	1	6			
	3 S $\rightarrow \bullet$ A1	5	16	4,5		
	4 A1 $\rightarrow \bullet$ b1 C1	4	12			
	5 A1 $\rightarrow \bullet$ a1	2	10			
1	6 S \rightarrow x \bullet C1 y1 Cn yn	13	27		y1	7
	7 C1 $\rightarrow \bullet$ An	7	18	8,9		
	8 An $\rightarrow \bullet$ bn Cn	8	19			
	9 An $\rightarrow \bullet$ an	9	23			
2	10 A1 \rightarrow a1 \bullet					
3	11 Start \rightarrow S \bullet \$	12	26			
4	12 A1 \rightarrow b1 \bullet C1	6	17	13		
	13 C1 $\rightarrow \bullet$ An	7	18	14,15		
	14 An $\rightarrow \bullet$ bn Cn	8	19			
	15 An $\rightarrow \bullet$ an	9	23			
	5	16 S \rightarrow A1 \bullet				
6	17 A1 \rightarrow b1 C1 \bullet					
7	18 C1 \rightarrow An \bullet					
8	19 An \rightarrow bn \bullet Cn	10	24	20		
	20 Cn $\rightarrow \bullet$ A1	11	25	21,22		
	21 A1 $\rightarrow \bullet$ b1 C1	4	12			
	22 A1 $\rightarrow \bullet$ a1	2	10			
	9	23 An \rightarrow an \bullet				
10	24 An \rightarrow bn Cn \bullet					
11	25 Cn \rightarrow A1 \bullet					
12	26 Start \rightarrow S \$ \bullet					
13	27 S \rightarrow x C1 \bullet y1 Cn yn	14	28			
14	28 S \rightarrow x C1 y1 \bullet Cn yn	15	32		yn	29
	29 Cn $\rightarrow \bullet$ A1	11	25	30,31		
	30 A1 $\rightarrow \bullet$ b1 C1	4	12			
	31 A1 $\rightarrow \bullet$ a1	2	10			
	15	32 S \rightarrow x C1 y1 Cn \bullet yn	16	33		
16	33 S \rightarrow x C1 y1 Cn yn \bullet					

Figure 6.31: LALR(1) analysis.

Item	Prop To	Initial	Pass 1	Pass 2
1	11	\$		
2	6	\$		
3	4,5,16	\$		
4	12		\$	
5	10		\$	
6	27		\$	
7	8,9,18	y1		
8	19		y1	
9	23		y1	
10			\$ y1 yn	
11	26		\$	
12	13,17		\$ y1 yn	
13	14,15,18		\$	y1 yn
14	19		\$	y1 yn
15	23		\$	y1 yn
16			\$	
17			\$	y1 yn
18			y1 \$	yn
19	20,24		y1 \$	yn
20	21,22,25		y1 \$	yn
21	12		y1 \$	yn
22	10		y1 \$	yn
23			y1 \$	yn
24			y1 \$	yn
25			y1 \$ yn	
26			\$	
27	28		\$	
28	32		\$	
29	25,30,31	yn		
30	12		yn	
31	10		yn	
32	33		\$	
33			\$	

Figure 6.32: Iterations for LALR(1) follow sets.

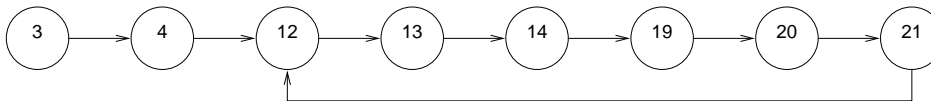


Figure 6.33: Embedded propagation subgraph.

1	Start	→	S \$
2	S	→	lp M rp
3			lb M rb
4			lp U rb
5			lb U rp
6	M	→	expr
7	U	→	expr

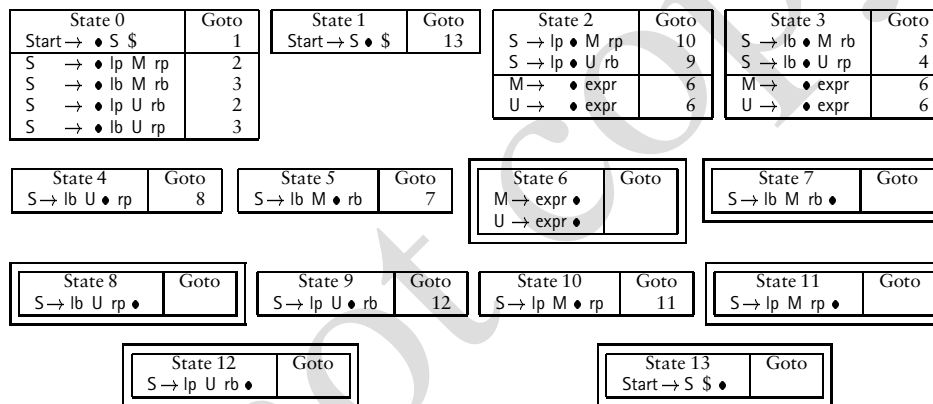
Figure 6.34: A grammar that is not LALR(k).

Figure 6.35: LR(0) construction.

$M \rightarrow \text{expr}, rp$ is sent to Item 8 and then to Item 14. Also, rb is sent to Item 12 and then to Item 14. Thus, $ItemFollow(14) = \{rb, rp\}$. Similarly, we compute $ItemFollow(15) = \{rb, rp\}$. Thus, State 6 contains a reduce/reduce conflict. For LALR(1), the rules $M \rightarrow \text{expr}$ and $U \rightarrow \text{expr}$ can each be followed by either rp or rb .

Because LALR(1) is based on LR(0), there is exactly one state with the kernel of State 6. Thus, States 2 and 3 must share State 6 when shifting an expr . If only we could split State 6, so that State 2 shifts to one version and State 3 shifts to the other, then the lookaheads in each state could resolve the conflict between $M \rightarrow \text{expr}$ and $U \rightarrow \text{expr}$. The LR(1) construction causes such splitting, because a state is uniquely identified not only by its kernel from LR(0) but also its lookahead information.

SLR(k) and LALR(k) supply information to LR(0) states to help resolve conflicts. In LR(k), such information is part of the items themselves. For LR(k), we extend an item's notation from $A \rightarrow \alpha \bullet \beta$ to $[A \rightarrow \alpha \bullet \beta, w]$. For LR(1), w is a (terminal) symbol that can follow A when this item becomes reducible. For LR(k), $k \geq 0$, w is a k -length string that can follow A after reduction. If symbols x and y can both

State	LR(0) Item	Goto State	Prop Edges Placed by Step		Initialize <i>ItemFollow</i>	
			27	29	First(γ)	28
0	1 $S \rightarrow \bullet S \$$	1	??		\$	2,3,4,5
	2 $S \rightarrow \bullet lp M rp$	2	6			
	3 $S \rightarrow \bullet lb M rb$	3	10			
	4 $S \rightarrow \bullet lp U rb$	2	7			
	5 $S \rightarrow \bullet lb U rp$	3	11			
2	6 $S \rightarrow lp \bullet M rp$	10	??		rp	8
	7 $S \rightarrow lp \bullet U rb$	9	??		rb	9
	8 $M \rightarrow \bullet expr$	6	14			
	9 $U \rightarrow \bullet expr$	6	15			
3	10 $S \rightarrow lb \bullet M rb$	5	??		rb	12
	11 $S \rightarrow lb \bullet U rp$	4	??		rp	13
	12 $M \rightarrow \bullet expr$	6	14			
	13 $U \rightarrow \bullet expr$	6	15			
6	14 $M \rightarrow expr \bullet$					
	15 $U \rightarrow expr \bullet$					

Figure 6.36: Partial LALR(1) analysis.

follow A when $A \rightarrow \alpha \bullet \beta$ becomes reducible, then the corresponding LR(1) state contains both $[A \rightarrow \alpha \bullet \beta, x]$ and $[A \rightarrow \alpha \bullet \beta, y]$.

Notice how nicely the notation for LR(k) generalizes LR(0). For LR(0), w must be a 0-length string. The only such string is λ , which provides no information at a possible point of reduction, since λ does not occur as input.

Examples of LR(1) items for the grammar in Figure 6.34 include $[S \rightarrow lp \bullet M rp, \$]$ and $[M \rightarrow expr \bullet, rp]$. The first item is not ready for reduction, but indicates that $\$$ will follow the reduction to S when the item becomes reducible in State 11. The second item calls for a reduction by rule $M \rightarrow expr$ when rp is the next input token. It is possible and likely that a given state contains several items that differ only in their follow symbol.

In LR(k), a state is a set of LR(k) items, and construction of the CFM is basically the same as with LR(0). States are represented by their kernel items, and new states are generated as needed. Figure 6.37 presents an LR(1) construction algorithm in terms of modifications to the LR(0) algorithm shown in Figures 6.8 and 6.9. At Step 31, any symbol that can follow B due to the presence of γ is considered; when $\gamma \Rightarrow^* \lambda$, then any symbol a that can follow A can also follow B . Thus, Step 31 considers each symbol in $First(\gamma a)$. The current state receives an item for each rule for B and each possible follow symbol. Figure 6.13 shows TRYRULEINSTATE—the LR(0) method for determining if a state calls for a particular

The following steps must be modified in Figures 6.8 and 6.9.

Step 7: We initialize *StartItems* by including LR(1) items that have \$ as the follow symbol:

$$\text{StartItems} \leftarrow \{ [\text{Start} \rightarrow \bullet \text{RHS}(p), \$] \mid p \in \text{PRODUCTIONSFOR}(\text{Start}) \}$$

Step 13: We augment the LR(0) item so that *ADVANCEDOT* returns the appropriate LR(1) items:

$$\text{return } (\{ [A \rightarrow \alpha \mathcal{X} \bullet \beta, a] \mid [A \rightarrow \alpha \bullet \mathcal{X} \beta, a] \in \text{state} \})$$

Step 15: This entire loop is replaced by the following:

```

foreach [ A → α • Bγ, a ] ∈ ans do
  foreach p ∈ PRODUCTIONSFOR(B) do
    foreach b ∈ First(γα) do
      ans ← ans ∪ { [ B → • RHS(p), b ] }

```

31

Figure 6.37: LR(1) construction.

```

procedure TRYRULEINSTATE(s, r)
  if [ LHS(r) → RHS(r) •, w ] ∈ s
  then call ASSERTENTRY(s, w, reduce r)
end

```

Figure 6.38: LR(1) version of TRYRULEINSTATE.

reduction. The LR(1) version of TRYRULEINSTATE is shown in Figure 6.38.

Figure 6.39 shows the LR(1) construction for the grammar in Figure 6.34. States 6 and 14 would be merged under LR(0). For LR(1), these states differ by the lookaheads associated with the reducible items. Thus, LR(1) is able to resolve what would have been a reduce/reduce conflict under LR(0).

The number of states such as States 6 and 14 that split during LR(1) construction is usually much larger. Instead of constructing a full LR(1) parse table, one could begin with LALR(1), which is based on the LR(0) construction. States could then be split selectively. As discussed in Exercise 34, LR(*k*) can resolve only the reduce/reduce conflicts that arise during LALR(*k*) construction. A shift/reduce conflict in LALR(*k*) will also be present in the corresponding LR(*k*) construction. Exercise 35 considers how to split LR(0) states on-demand, in response to reduce/reduce conflicts that arise in LALR(*k*) constructions.

Summary

This concludes our study of LR parsers. We have investigated a number of LR table-building methods, from LR(0) to LR(1). The intermediate methods—SLR(1) and LALR(1)—are the most practical. In particular, LALR(1) provides excellent conflict

State	GOTO
State 0 [Start → • S \$, \$]	1
[S → • lp M rp , \$]	2
[S → • lb M rb , \$]	3
[S → • lp U rb , \$]	2
[S → • lb U rp , \$]	3
State 1 [Start → S • \$, \$]	13
State 2 [S → lp • M rp , \$]	10
[S → lp • U rb , \$]	9
[M → • expr , rp]	6
[U → • expr , rb]	6
State 3 [S → lb • M rb , \$]	5
[S → lb • U rp , \$]	4
[M → • expr , rb]	14
[U → • expr , rp]	14
State 4 [S → lb U • rp , \$]	8
State 5 [S → lb M • rb , \$]	7
State 6 [M → expr • , rp]	
[U → expr • , rb]	
State 7 [S → lb M rb • , \$]	
State 8 [S → lb U rp • , \$]	
State 9 [S → lp U • rb , \$]	12
State 10 [S → lp M • rp , \$]	11
State 11 [S → lp M rp • , \$]	
State 12 [S → lp U rb • , \$]	
State 13 [Start → S \$ • , \$]	
State 14 [M → expr • , rb]	
[U → expr • , rp]	

Figure 6.39: LR(1) construction.

resolution and generates very compact tables. Tools based on LALR(1) grammars are discussed in Chapter Chapter:global:seven. Such tools are indispensable for language modification and extension. Changes can be prototyped using an LALR(1) grammar for the language's syntax. When conflicts occur, the methods discussed in Section 6.4 help identify why the proposed modification may not work. Because of their efficiency and power, LALR(1) grammars are available for most modern programming languages. Indeed, the syntax of modern programming languages is commonly designed with LALR(1) parsing in mind.

Exercises

- Build the CFSM and the parse table for the grammar shown in Figure 6.1.
- Using the knitting analogy of Section 6.2.2, show the sequence of LR shift and reduce actions for the grammar of Figure 6.1 on the following strings.
 - plus plus num num num \$
 - plus num plus num num \$
- Figure 6.6 traces a bottom-up parse of an input string using the table shown in Figure 6.5. Trace the parse of the following strings.
 - q \$
 - c \$
 - a d c \$
- Build the CFSM for the following grammar.

1	Prog	→	Block \$
2	Block	→	begin StmtList end
3	StmtList	→	StmtList semi Stmt
4			Stmt
5	Stmt	→	Block
6			Var assign Expr
7	Var	→	id
8			id lb Expr rb
9	Expr	→	Expr plus T
10			T
11	T	→	Var
12			lp Expr rp

- Show the LR parse table for the CFSM constructed in Exercise 4.
- Which of following grammars are LR(0)? Explain why.

(a)	1	S	→	StmtList \$
	2	StmtList	→	StmtList semi Stmt
	3			Stmt
	4	Stmt	→	s

(b)	1	S	→	StmtList \$
	2	StmtList	→	Stmt semi StmtList
	3			Stmt
	4	Stmt	→	s

(c)

1	S	→	StmtList \$
2	StmtList	→	StmtList semi StmtList
3			Stmt
4	Stmt	→	s

(d)

1	S	→	StmtList \$
2	StmtList	→	s StTail
3	StTail	→	semi StTail
4			λ

7. Show that the CFSM corresponding to a LL(1) grammar has the following property. Each state has exactly one kernel item if the grammar is λ -free.
8. Prove or disprove that all λ -free LL(1) grammars are LR(0).
9. Explain why the language defined by the following grammar is *inherently nondeterministic*—there is no LALR(k) grammar for this language.

1	Start	→	Single a
2			Double b
3	Single	→	0 Single 1
4			0 1
5	Double	→	0 Double 1 1
6			0 1 1

10. Given the claim of Exercise 9, explain why the following statement is true or false. There is no LR(k) grammar for the language

$$\{0^n 1^n a\} \cup \{0^n 1^{2n} b\}.$$
11. Discuss why is it not possible during LR(0) construction to obtain a shift/reduce conflict on a nonterminal.
12. Discuss why there cannot be an unambiguous CFG for the language

$$\{a^i b^j c^k \mid i = j \text{ or } j = k; i, j, k \geq 1\}.$$
13. Complete the LR(0) construction for the grammar in Figure 6.18.
14. Show that LL(1) construction fails for an unambiguous grammar that is not LR(1).
15. Show that the grammar in Figure 6.19 is LR(0).

16. Complete the LR(0) construction for the grammar shown in Figure 6.24. Your state numbers should agree with those shown in the partial LR(0) construction.

17. Show the LR(0) construction for the following grammars.

(a)

1	Start	→	S \$
2	S	→	id assign E semi
3	E	→	E plus P
4			P
5	P	→	id
6			lp E rp
7			id assign E

(b)

1	Start	→	S \$
2	S	→	id assign A semi
3	A	→	id assign A
4			E
5	E	→	E plus P
6			P
7	P	→	id
8			lp A rp

(c)

1	Start	→	S \$
2	S	→	id assign A semi
3	A	→	id assign A
4			E
5	E	→	E plus P
6			P
7			P plus
8	P	→	id
9			lp A rp

(d)

1	Start	→	S \$
2	S	→	id assign A semi
3	A	→	Pre E
4	Pre	→	Pre id assign
5			λ
6	E	→	E plus P
7			P
8	P	→	id
9			lp A rp

(e)

1	Start	→	S \$
2	S	→	id assign A semi
3	A	→	Pre E
4	Pre	→	id assign Pre
5			λ
6	E	→	E plus P
7			P
8	A	→	id
9			lp A rp

(f)

1	Start	→	S \$
2	S	→	id assign A semi
3	A	→	id assign A
4			E
5	E	→	E plus P
6			P
7	P	→	id
8			lp A semi A rp
9			lp V comma V rp
10			lb A comma A rb
11			lb V semi V rb
12	V	→	id

(g)

1	Start	→	S \$
2	S	→	id assign A semi
3	A	→	id assign A
4			E
5	E	→	E plus P
6			P
7	P	→	id
8			lp id semi id rp
9			lp A rp

18. Which of the grammars in Exercise 17 are LR(0)? Justify your answers.
19. Complete the SLR(1) construction for the grammar shown in Figure 6.25. Show the resulting parse table.
20. Extend the grammar given in Figure 6.20 to accommodate standard expressions involving addition, subtraction, multiplication, and division. Model the syntax and semantics for these operators according to Java or C.
21. Extend the grammar as directed in Exercise 20, but introduce an exponentiation operator that is *right-associating*. Let the operator (denoted by “ \ast ”) have the highest priority, so that the value of $3 + 4 \times 5 \ast 2$ is 103.

22. Repeat Exercise 21 but add the ability to enclose expressions with parentheses, to control how expressions are grouped together. Thus, the value of $((3 + 4) \times 5) \star 2$ is 1225.

23. Which of the grammars in Exercise 17 are SLR(1)? Justify your answers.

24. Generalize the algorithm given in Section 6.5.1 from SLR(1) to SLR(k).

25. Show that

(a) For any state s containing the item $A \rightarrow \alpha \bullet \beta$, $ItemFollow((s, A \rightarrow \alpha \bullet \beta)) \subseteq Follow(A)$

(b)

$$\bigcup_s \bigcup_{A \rightarrow \alpha_i \bullet \beta_i \in s} ItemFollow((s, A \rightarrow \alpha_i \bullet \beta_i)) = Follow(A)$$

26. Perform the LALR(1) construction for the following grammar:

1	Start	→	S \$
2	S	→	x C1 y1 C2 y2 C3 y3
3			A1
4	A1	→	b1 C1
5			a1
6	A2	→	b2 C2
7			a2
8	A3	→	b3 C3
9			a3
10	C1	→	A2
11	C2	→	A1
12			A3
13	C3	→	A2

27. Using the grammars in Figure 6.30 and Exercise 26 as a guide, show how to generate a grammar that requires n iterations for LALR(1) convergence.

28. For the grammar shown in Figure 6.34, complete the LALR(1) construction from Figure 6.36.

29. Which of the grammars in Exercise 17 are LALR(1)? Justify your answers.

30. Show the LR(1) construction for the grammar in Exercise 4.

31. Define the **quasi-identical states** of an LR(1) parsing machine to be those states whose kernel productions are identical. Such states are distinguished only by the lookahead symbols associated with their productions. Given the LR(1) machine built for Exercise 30, do the following.

- (a) List the quasi-identical states of the LR(1) machine.
 (b) Merge each set of quasi-identical states to obtain an LALR(1) machine.
32. Starting with the CFSM built in Exercise 4, compute the LALR(1) lookahead information. Compare the resulting LALR(1) machine with the machine obtained in Exercise 31
33. Which of the grammars in Exercise 17 are LR(1)? Justify your answers.
34. Consider a grammar G and its LALR(1) construction. Suppose that a shift/reduce conflict occurs in G 's LALR(1) construction. Prove that G 's LR(1) construction also contains a shift/reduce conflict.
35. Describe an algorithm that computes LALR(1) and then splits states as needed in an attempt to address conflicts. Take note of the issue raised in Exercise 34.
36. Using a grammar for the C programming language, try to extend the syntax to allow *nested* function definitions. For example, you might allow function definitions to occur inside any compound statement.
 Report on any difficulties you encounter, and discuss possible solutions. Justify the particular solution you adopt.
37. Using a grammar for the C programming language, try to extend the syntax so that a compound statement can appear to compute a value. In other words, allow a compound statement to appear wherever a simple constant or identifier could appear. Semantically, the value of a compound statement could be the value associated with its last statement.
 Report on any difficulties you encounter, and discuss possible solutions. Justify the particular solution you adopt.
38. In Figure 6.3, Step 2 pushes a state on the parse stack. In the bottom-up parse shown in Figure 6.6, stack cells show both the state and the input symbol causing the state's shift onto the stack. Explain why the input symbol's presence in the stack cell is superfluous.
39. Recall the **dangling else** problem introduced in Chapter Chapter:global:five. Following is a grammar for a simplified language that allows conditional statements.

```

1 Start  → Stmt $
2 Stmt  → if e then Stmt else Stmt
3       | if e then Stmt
4       | other
  
```

Explain why the following grammar is or is not LALR(1).

40. Consider the following grammar.

1	Start	→	Stmt \$
2	Stmt	→	Matched
3			Unmatched
4	Matched	→	if e then Matched else Matched
5			other
6	Unmatched	→	if e then Matched else Unmatched
7			if e then Unmatched

- (a) Explain why the grammar is or is not LALR(1).
 (b) Is the language of this grammar the same as the language of the grammar in Exercise 39? Why or why not?

41. Repeat Exercise 40, adding the production $\text{Unmatched} \rightarrow \text{other}$ to the grammar.

42. Consider the following grammar.

1	Start	→	Stmt \$
2	Stmt	→	Matched
3			Unmatched
4	Matched	→	if e then Matched else Matched
5			other
6	Unmatched	→	if e then Matched else Unmatched
7			if e then Stmt

- (a) Explain why the grammar is or is not LALR(1).
 (b) Is the language of this grammar the same as the language of the grammar in Exercise 39? Why or why not?

43. Based on the material in Exercises 39, 40, and 42, construct an LALR(1) grammar for the language defined by the following grammar.

1	Start	→	Stmt \$
2	Stmt	→	if e then Stmt else Stmt
3			if e then Stmt
4			while e Stmt
5			repeat Stmt until e
6			other

44. Show that there exist non-LL(1) grammars that are

- (a) LR(0)
 (b) SLR(1)
 (c) LALR(1)

45. Normally, an LR parser traces a rightmost derivation (in reverse).

- (a) How could an LR parser be modified to produce a leftmost parse as LL(1) parsers do? Describe your answer in terms of the algorithm in Figure 6.3.
- (b) Would it help if we knew that the LR table was constructed for an LL grammar?
46. For each of the following, construct an appropriate grammar.
- The grammar is SLR(3) but not SLR(2).
 - The grammar is LALR(2) but not LALR(1).
 - The grammar is LR(2) but not LR(1).
 - The grammar is LALR(1) and SLR(2) but not SLR(1).
47. Construct a single grammar that has *all* of the following properties.
- It is SLR(3) but not SLR(2).
 - It is LALR(2) but not LALR(1).
 - It is LR(1).
48. For every $k > 1$ show that there exist grammars that are SLR($k + 1$), LALR($k + 1$), and LR($k + 1$) but not SLR(k), LALR(k), or LR(k).
49. Consider the grammar generated by $1 \leq i, j \leq n$, $i \neq j$ using the following template.

$$\begin{array}{lcl} S & \rightarrow & X_i z_i \\ X_i & \rightarrow & y_j X_i \\ & & | y_j \end{array}$$

The resulting grammar has $O(n^2)$ productions.

- Show that the CFSM for this grammar has $O(2^n)$ states.
- Is the grammar SLR(1)?

7

Syntax-Directed Translation

The parsers discussed in Chapters Chapter:global:five and Chapter:global:six can recognize syntactically valid inputs. However, compilers are typically required to perform some translation of the input source into a target representation, as discussed in Chapter Chapter:global:two. Some compilers are *single-pass*, translating programs as they parse without taking any intermediate steps. Most compilers accomplish translation using multiple passes. Instead of repeatedly scanning the input program, compilers typically create an intermediate structure called the **abstract syntax tree** (AST) as a by-product of the parse. The AST then serves as a mechanism for conveying information between compiler passes. In this chapter we study how to formulate grammars and production-specific code sequences for the purpose of direct translation or the creation of ASTs. Sections 7.3 and 7.4 consider bottom-up and top-down translation, respectively; Section 7.5 considers the design and synthesis of ASTs.

7.1 Overview

The work performed by a compiler while parsing is generally termed *syntax-directed translation*. The grammar on which the parser is based causes a specific sequence of derivation steps to be taken for a given input program. In constructing the derivation, a parser performs a sequence of *syntactic actions* as described in Chapters Chapter:global:five and Chapter:global:six; such actions (*e.g.*, *shift* and *reduce* for LR parsing) pertain only with the grammar's terminal and nonterminal symbols. To achieve syntax-directed translation, most parser generators allow a segment of

computer code to be associated with each grammar production; such code is executed whenever the production participates in the derivation. Typically, the code deals with the *meaning* of the grammar symbols. For example, the syntactic token identifier has a specific value when a production involving identifier is applied. The *semantic actions* performed by the parser through execution of the supplied code segment can reference the actual string comprising the terminal, perhaps for the purpose of generating code to load or store the value associated with the identifier.

In automatically generated parsers, the parser driver (Figure Figure:six:driver for LR parsing) is usually responsible for executing the semantic actions. To simplify calling conventions, the driver and the grammar's semantic actions are written in the same programming language. Parser generators also provide a mechanism for the supplied semantic actions to reference semantic values associated with the associated production's grammar symbols. Semantic actions are also easily inserted into *ad hoc* parsers by specifying code sequences that execute in concert with the parser.

Formulating an appropriate set of semantic actions requires a firm understanding of how derivations are traced in a parse, be it bottom-up or top-down. When one encounters difficulties in writing clear and elegant semantic actions, grammar reformulation can often simplify the task at hand by computing semantic values at more appropriate points. Even after much work is expended to obtain an LALR(1) grammar for a programming language, it is not unusual to revise the grammar during this phase of compiler construction.

7.2 Synthesized and inherited attributes

In a parse (derivation) tree, each node corresponds to a grammar symbol utilized in some derivation step. The information associated with a grammar symbol by semantic actions become *attributes* of the parse tree. In Section 7.3 we examine how to specify semantic actions for bottom-up (LALR(1)) parsers, which essentially create parse trees in a *postorder* fashion. For syntax-directed translation, this style nicely accommodates situations in which attributes primarily flow from the leaves of a derivation tree toward its root. If we imagine that each node of a parse tree can consume and produce information, then nodes consume information from their children and produce information for their parent in a bottom-up parse. An example using such *synthetic attributes* flow is shown in Figure 7.1: the attribute associated with each node is an expression value, and the value of the entire computation becomes available at the root of the parse tree.

By contrast, consider the problem of propagating symbol table information through a parse tree, so that inner scopes have access to information present in outer scopes. As shown in Figure 7.1, such *inherited attributes* flow from the root of the parse tree toward its leaves. As discussed in Section 7.4, top-down parsing nicely accommodates such information flow because its parse trees are created in a preorder fashion.

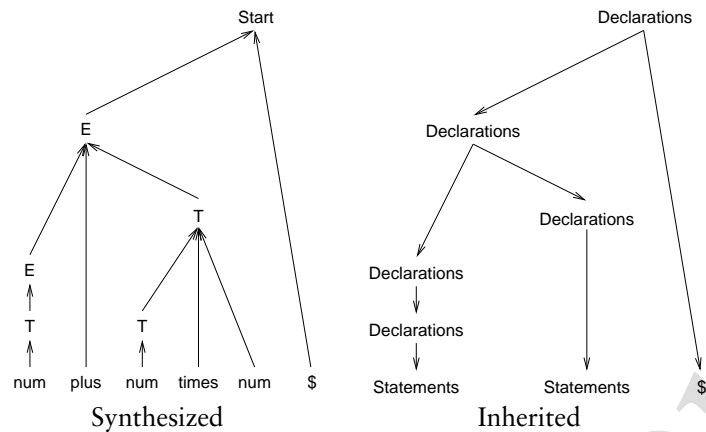


Figure 7.1: Synthesized and inherited attributes.

Most programming languages require information flow in both directions. While some tools allow both styles of propagation to coexist peacefully, such tools are not in widespread use; moreover, choreographing attribute flow can be difficult to specify and maintain. In the ensuing sections we focus only on syntax-directed parsers that allow attributes to be synthesized or inherited, but not both. In practice, this is not a serious limitation. attributes flowing in the opposite direction are typically accommodated by global structures. For example, symbol information can be stored in a global *symbol table* in a parser based on synthesized attributes.

7.3 Bottom-up translation

In this section we consider how to incorporate semantic actions into bottom-up parsers. Such parsers are almost always generated automatically by tools such as Cup, Yacc, or Bison, which allow specification of code sequences that execute as reductions are performed.

Consider an LR parser that is about to perform a reduction using the rule $A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_n$. As discussed in Chapter Chapter:global:six, the symbols $\mathcal{X}_1 \dots \mathcal{X}_n$ are on top-of-stack prior to the reduction, with \mathcal{X}_n topmost. The n symbols are popped from the stack and the symbol A is pushed on the stack. In such a bottom-up parse, it is possible that previous reductions have associated semantic information with the symbols $\mathcal{X}_1 \dots \mathcal{X}_n$. The semantic action associated with $A \rightarrow \mathcal{X}_1 \dots \mathcal{X}_n$ can take such information into consideration in synthesizing the semantic information to be associated with A . In other words, the bottom-up parser operates two stacks: the *syntactic stack* manipulates terminal and nonterminal symbols as described in Chapter Chapter:global:six, while the *semantic stack* manipulates values associated with the symbols. The code for maintaining both stacks is generated

- 1 Start \rightarrow Digs_{ans} \$
 call PRINT(*ans*)
- 2 Digs_{up} \rightarrow Digs_{below} d_{next}
 $up \leftarrow below \times 10 + next$
- 3 | d_{first}
 $up \leftarrow first$

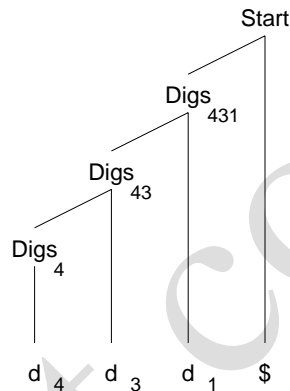


Figure 7.2: Possible grammar and parse tree for decimal 4 3 1 \$.

automatically by the parser generator, based on the grammar and semantic-action code provided by the compiler writer.

7.3.1 Left recursive rules

To illustrate a bottom-up style of translation, we begin with the grammar shown in Figure 7.2. The translation task consists of computing the value of a string of digits, base 10. Compilers normally process such strings during lexical analysis, as described in Chapter Chapter:global:three. Our example serves pedagogically to illustrate the problems that can arise with syntax-directed translation during a bottom-up parse.

Syntactically, the terminal *d* represents a single digit; the nonterminal *Digs* generates one or more such symbols. Some of the grammar symbols are annotated with tags, as in *Digs_{up}*. The symbol *Digs* is a grammar terminal or nonterminal symbol; the symbol *up* represents the semantic value associated with the symbol. Such values are provided by the scanner for terminal symbols; for nonterminals, the grammar's semantic actions establish the values. For example, Rule 3 contains *d_{first}*. The syntactic element *d* represents an integer, as discovered by the scanner; the semantic tag *first* represents the integer's value, also provided by the scanner. Parser generators offer methods for declaring the *type* of the semantic symbols; for

the sake of language-neutrality and clarity we omit type declarations in our examples. In the grammar of Figure 7.2, all semantic tags would be declared to have integer value.

We now analyze how the semantic actions in Figure 7.2 compute the base-10 value of the digit string. To appreciate the interaction of Rules 2 and 3, we examine the order in which a bottom-up parse applies these rules. Figure 7.2 shows a parse tree for the input string 4 3 1 \$. In Chapter Chapter:global:six we learned that a bottom-up parse traces a rightmost derivation in reverse. The rules for Digs are applied as follows:

Digs \rightarrow d This rule must be applied first; d corresponds to the input digit 4. The semantic action causes the value of the first digit (4) to be transferred as the semantic value for Digs. Semantic actions such as these are often called *copy rules* because they serve only to propagate values up the parse tree.

Digs \rightarrow Digs d Each subsequent d is processed by this rule. The semantic action $up \leftarrow below \times 10 + next$ multiplies the value computed thus far (*below*) by 10 and then adds in the semantic value of the current d (*next*).

As seen in the above example, left-recursive rules are amenable to semantic processing of input text from left to right. Exercise 1 considers the difficulty of computing a digit string's value when the grammar rule is right-recursive.

7.3.2 Rule cloning

We extend our language slightly, so that a string of digits can be prefaced by an x, in which case the digit string should be interpreted base-8 rather than base-10; any input digit out of range for a base-8 number is ignored. For this task we propose the grammar shown in Figure 7.3. Rule 3 generates strings of digits as before; Rule 2 generates such strings prefaced with an x. Prior to inserting semantic actions, we examine the parse tree shown in Figure 7.3. As was the case with our previous example, the first d symbol is processed first by Rule 4 and the remaining d symbols are processed by Rule 5. If the string is to be interpreted base-8, then the semantic action for Rule 5 should multiply the number passed up the parse tree by 8; otherwise, 10 should be used at the multiplier.

The problem with the parse tree shown in Figure 7.3 is that the x is shifted on the stack with no possible action to perform: actions are possible only on reductions. Thus, when the semantic actions for Rule 5 execute, it is unknown whether we are processing a base-10 or base-8 number.

There are several approaches to remedy this problem, but the one we consider first involves *cloning* productions in the grammar. We construct two kinds of digit strings, one derived from OctDigs and the other derived from DecDigs, to obtain the grammar and semantic actions shown in Figure 7.4. Rules 4 and 5 interpret strings of digits base-10; Rules 6 and 7 generate the same syntactic strings but interpret

```

1  Start  →  Num $
2  Num    →  x Digs
3         |  Digs
4  Digs   →  Digs d
5         |  d

```

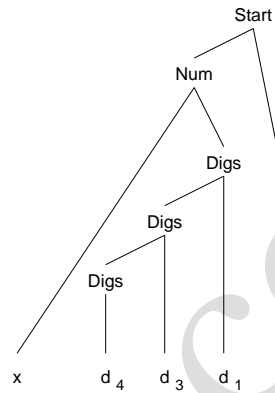


Figure 7.3: Possible grammar and parse tree for octal x 4 3 1 \$.

```

1  Start      →  Numans $
                   call PRINT( ans )
2  Numans    →  x OctDigsoctans
                   ans ← octans
3         |  DecDigsdecans
                   ans ← decans
4  DecDigsup →  DecDigsbelow dnext
                   up ← below × 10 + next
5         |  dfirst
                   up ← first
6  OctDigsup →  OctDigsbelow dnext
                   up ← below × 8 + next
7         |  dfirst
                   up ← first

```

Figure 7.4: Grammar with cloned productions.

1	Start	→	Num_{ans} \$ $\text{call PRINT}(ans)$
2	Num_{ans}	→	$\text{SignalOctal Digs}_{octans}$ $ans \leftarrow octans$
3			$\text{SignalDecimal Digs}_{decans}$ $ans \leftarrow decans$
4	SignalOctal	→	x $base \leftarrow 8$
5	SignalDecimal	→	λ $base \leftarrow 10$
6	Digs_{up}	→	$\text{Digs}_{below} d_{next}$ $up \leftarrow below \times base + next$
7			d_{first} $up \leftarrow first$

Figure 7.5: Use of λ -rules to take semantic action.

their meaning base-8. The semantic check for octal digits is left as Exercise 2. With this example, we see that grammars are modified for reasons other than the parsing issues raised in Chapters Chapter:global:four and Chapter:global:five. Often, a translation task can become significantly easier if the grammar can be modified to accommodate efficient flow of semantic information.

If the grammars in Figures 7.3 and 7.4 were examined without their semantic actions, the grammar in Figure 7.3 would be favored because of its simplicity. Insertion of semantic actions often involves inflating a grammar with productions that are not necessary from a syntactic point of view. These extra productions allow needed flexibility in accomplishing semantic actions at appropriate points in the parse. However, grammar modifications can affect the parsability of a grammar. Such modifications should be made in small increments, so that the compiler writer can respond quickly and accurately to any parsability problems raised by the parser generator.

7.3.3 Global variables and λ -rules

We now examine an alternative for the grammar shown in Figure 7.3. Recall that the x was shifted without semantic notice: actions are performed only at reductions. We can modify the grammar from Figure 7.3 so that a reduction takes place after the x has been shifted but prior to processing any of the digits. The resulting grammar is shown in Figure 7.5. This grammar avoids copying the productions that generate a string of digits; however, the grammar assigns and

references a global variable (*base*) as a *side-effect* of processing an *x* (Rule 4) or nothing (Rule 5).

In crafting a grammar with appropriate semantic actions, care must be taken to avoid changing the actual language. Although octal strings are preceded by an *x*, decimal strings are unheralded in this language; the λ -rule present in Figure 7.5 serves only to cause a reduction when an *x* has *not* been seen. This semantic action is required to initialize *base* to its default value of 10. If the λ -rule were absent and *base* were initialized prior to operating the parser, then the parser would not correctly parse a base-8 number followed by a base-10 number, because *base* would have the residual value of 8.

While they can be quite useful, global variables are typically considered undesirable because they can render a grammar difficult to write and maintain. Synthesized attributes, such as the digits that are pushed up a parse tree, can be localized with regard to a few productions; on the other hand, global variables can be referenced in *any* semantic action. However, programming language constructs that lie outside the context-free languages must be accommodated using global variables; symbol tables (Chapter Chapter:global:eight) are the most common structure deserving of global scope.

7.3.4 Aggressive grammar restructuring

We expand our digits example further, so that a single-digit base is supplied along with the introductory *x*. Some examples of legal inputs and their interpretation are as follows:

Input	Meaning	Value (base-10)
4 3 1 \$	431_{10}	431
x 8 4 3 1 \$	431_8	281
x 5 4 3 1 \$	431_5	116

A grammar for the new language is shown in Figure 7.6; semantic actions are included that use the global variable *base* to enforce the correct interpretation of digit strings.

While the solution in Figure 7.6 suffices, there is a solution that avoids use of global variables. The grammar as given does not place information in the parse tree where it is useful for semantic actions. This is a common problem, and the following steps are suggested as a mechanism for obtaining a grammar more appropriate for bottom-up, syntax-directed translation:

1. Sketch the parse tree that would allow bottom-up synthesis and translation without use of global variables.
2. Revise the grammar to achieve the desired parse tree.
3. Verify that the revised grammar is still suitable for parser construction. For example, if Cup or Yacc must process the grammar, then the grammar should remain LALR(1) after revision.

```

1  Start      → Numans $
                   call PRINT(ans)
2  Numans    → x SetBase Digsbaseans
                   ans ← baseans
3                   | SetBaseTen Digsdecans
                   ans ← decans
4  SetBase    → dval
                   base ← val
5  SetBaseTen → λ
                   base ← 10
6  Digsup    → Digsbelow dnext
                   up ← below × base + next
7                   | dfirst
                   up ← first

```

Figure 7.6: Strings with an optionally specified base.

4. Verify that the grammar still generates the same language, using proof techniques or rigorous testing.

The last step is the trickiest, because it is easy to miss subtle variations among grammars and languages.

For the problem at hand, we can avoid a global *base* variable if we can process the base early and have it propagate up the parse tree along with the value of the processed input string. Figure 7.7 sketches the parse tree we desire for the input $x\ 5\ 4\ 3\ 1\ \$$. The x and the first d , which specifies the base are processed in the triangle; the base can then propagate up the tree and participate in the semantic actions. From this tree we rewrite the rules for *Digs* to obtain the grammar shown in Figure 7.8.

The grammar in Figure 7.8 is fairly concise but it involves one novelty: the semantic value for *Digs* consists of two components

val which contains the interpreted value of the processed input string

base which conveys the base for interpretation

The semantic actions for Rule 3 serve to copy the base from its inception at Rule 2.

Experienced compiler writers make frequent use of rich semantic types and grammar restructuring to localize the effects of semantic actions. The resulting parsers are (mostly) free of global variables and easier to understand and modify.

Unfortunately, our grammar modification is deficient because it involves a subtle change in the language; this point is explored in great detail in Exercise 3.

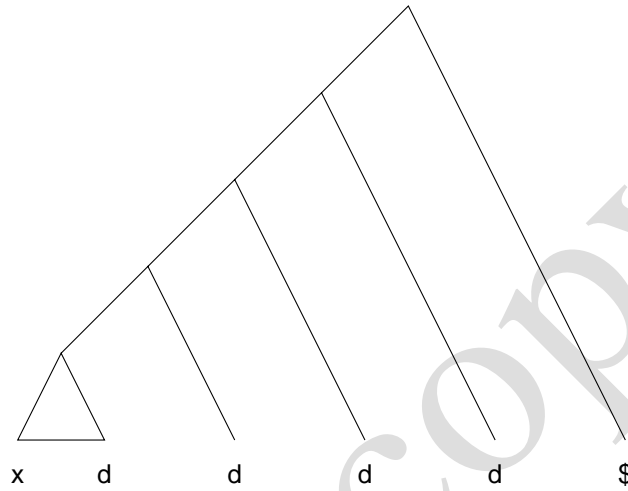


Figure 7.7: Desired parse tree.

```

1 Start → Digsans $
          call PRINT(ans.val)
2 Digsup → SetBasebasespec
          up.base ← basespec
          up.val ← 0
3       | Digsbelow dnext
          up.val ← below.val × below.base + next
          up.base ← below.base
4 Setbasen → λ
          n ← 10
5       | x dnum
          n ← num

```

Figure 7.8: Grammar that avoids global variables.

1	Start	→	Value \$
2	Value	→	num
3			lp Expr rp
4	Expr	→	plus Value Value
5			prod Values
6	Values	→	Value Values
7			λ

Figure 7.9: Grammar for Lisp-like expressions.

7.4 Top-down translation

In this section we discuss how semantic actions can be performed in top-down parsers. Such parsers are typically generated by hand in the *recursive-descent* style discussed in Chapters Chapter:global:two and Chapter:global:five. The result of such parser construction is a program; semantic actions are accomplished by code sequences inserted directly into the parser.

To illustrate this style of translation, we consider the processing of Lisp-like expressions, defined by the grammar shown in Figure 7.9. Rule 1 generates an outermost expression whose value should be printed by the semantic actions. A Value is defined by Rules 2 and 3, which generate an atomic value via a num or a parenthesized expression, respectively. Rules 4 and 5 generate a the sum of two values and a product of zero or more values, respectively. Following are some observations about this grammar.

- This disparate treatment of addition (which takes two operands) and multiplication (which takes arbitrary operands) is for pedagogical purposes; it would otherwise be considered poor language design.
- A more complete expression grammar is considered in Exercises 5 and 6.
- The recursive rule for Values is right-recursive to accommodate top-down parsing.

A recursive-descent parser for the grammar in Figure 7.9 is shown in Figure 7.10. As discussed in Chapter Chapter:global:five, the parser contains a procedure for each nonterminal in the grammar. To conserve space, the error checks normally present at the end of each **switch** statement are absent in Figure 7.10. If top-down parsers applied semantic actions only when a derivation step is taken, then information could only be passed from the root of a parse tree toward its leaves. To evaluate programs written in the language of Figure 7.9, the tree must be evaluated bottom-up. It is common in recursive descent parsing to allow semantic actions both before and after a rule's right-hand side has been discovered. Of course, in an *ad hoc* parser, code can be inserted anywhere.


```

procedure START(ts)
  switch ()
    case ts.PEEK() ∈ { num, lp }
      ans ← VALUE()
      call MATCH(ts, $)
      call PRINT(ans)
    end
  function VALUE(ts) : int
    switch ()
      case ts.PEEK() ∈ { num }
        call MATCH(ts, num)
        ans ← num.VALUEOF()
        return (ans)
      case ts.PEEK() ∈ { lp }
        call MATCH(ts, lp)
        ans ← EXPR()
        call MATCH(ts, rp)
        return (ans)
    end
  function EXPR(ts) : int
    switch ()
      case ts.PEEK() ∈ { plus }
        call MATCH(ts, plus)
        op1 ← VALUE()           1
        op2 ← VALUE()           2
        return (op1 + op2)     3
      case ts.PEEK() ∈ { prod }
        call MATCH(ts, prod)
        ans ← VALUES()         4
        return (ans)
    end
  function VALUES(ts) : int
    switch ()
      case ts.PEEK() ∈ { num, lp }
        factor ← VALUE()
        product ← VALUES()
        return (factor × product)  5
      case ts.PEEK() ∈ { rp }
        return (1)                6
    end

```

Figure 7.10: Recursive-descent parser with semantic actions.

In bottom-up parsing, semantic values are associated with grammar symbols. This effect can be accomplished in a recursive-descent parser by instrumenting the parsing procedures to return semantic values. The code associated with semantically active nonterminal symbols is modified to return an appropriate semantic value as follows.

- In Figure 7.10, each method except START is responsible for computing a value associated with the string of symbols it derives; these methods are therefore instrumented to return an integer.
- The code for Rule 4 captures the values of its two operands at Steps 1 and 2, returning their sum at Step 3.
- The code at Step 4 anticipates that VALUES will return the product of its expressions.
- Because Values participates only in Rule 5, the semantic actions in VALUES can be customized to form such a product. Exercise 5 considers accommodation of functionally separate distinct uses of the nonterminal Values.
- The most subtle aspect of the semantic actions concerns the formation of a product of values in VALUES. With reference to the parse tree shown in Figure 7.11, a leftmost derivation generates the Value symbols from left to right. However, the semantic actions compute values *after* a rule has been applied, in the style of an LR bottom-up parse. Thus, the invocation of VALUES that finishes first is the one corresponding to $\text{Values} \rightarrow \lambda$; this rule seeds the product at Step 6. The semantic actions then progress up the tree applying the code at Step 5, incorporating Value expressions from right to left.

7.5 Abstract Syntax Trees

While most of a compiler's tasks can be performed in a single pass via syntax-directed translation, modern software practice frowns upon delegating so much functionality to a single component such as the parser. Tasks such as semantic analysis, symbol table construction, program optimization, and code generation are each deserving of separate treatment in a compiler; squeezing them into a single compiler pass is an admirable feat of engineering, but the resulting compiler is difficult to extend or maintain.

In designing multipass compilers, considerable attention should be devoted to designing the compiler's **intermediate representation** (IR): the form that serves to communicate information between compiler passes. In some systems, the source language itself serves as the IR; such *source-to-source* compilers are useful in research and educational settings, where the effects of a compiler's restructuring transformations (Chapter Chapter:global:sixteen) can be easily seen. Such systems

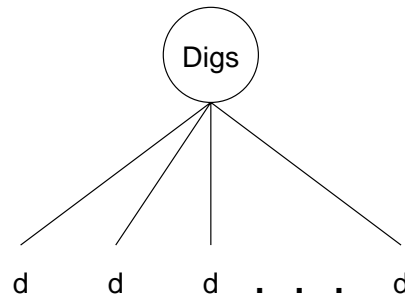


Figure 7.12: Abstract syntax tree for Digs.

no grammar that can generate such a concrete syntax tree directly: each production has a fixed number of symbols on its right-hand side. However, the tree in Figure 7.12 captures the order in which the `d` symbols appear, which suffices to translate the sequence of symbols into a number.

Because the AST is centrally involved in most of a compiler's activities, its design can and should evolve with the functionality of the compiler. A properly designed AST simplifies both the work required in a single compiler pass as well as the manner in which the compiler passes communicate. Given a source programming language `L`, the development of a grammar and an AST typically proceeds as follows:

1. An unambiguous grammar for `L` is devised. The grammar may contain productions that serve to disambiguate the grammar. The grammar is proved unambiguous through successful construction of an LR or LL parser.
2. An AST for `L` is devised. The AST design typically elides grammar details concerned with disambiguation. Semantically useless symbols such as “,” and “;” are also omitted.
3. Semantic actions are placed in the grammar to construct the AST. The design of the AST may require grammar modifications to simplify or localize construction. The semantic actions often rely upon utility methods for creating and manipulating AST components.
4. Passes of the compiler are designed. Each pass may place new requirements on the AST in terms of its structure and contents; grammar and AST redesign may then be desirable.

To illustrate the process of AST design, we consider the language Cicada, which offers *constants*, *identifiers*, *conditionals*, *assignment*, *dereferences*, and *arithmetic expressions*. While this language is relatively simple, its components are found in most programming languages; moreover, issues that arise in real AST design are present in Cicada.

1	Start	→	Stmt \$
2	Stmt	→	L assign E
3			E
4			if (E) Stmt fi
5			if (E) Stmt else Stmt fi
6	L	→	id
7			deref R
8	R	→	L
9			num
10			addr L
11	E	→	E plus T
12			T
13	T	→	T times F
14			F
15	F	→	(E)
16			R

Figure 7.13: Infix assignment and expression grammar.

7.5.1 Devising an unambiguous grammar.

The defining grammar for Cicada is shown in Figure 7.13. This grammar is already unambiguous and is suitable for LALR(1); however, it is not suitable for LL(1) parsing because of its left-recursion. Strings derived from the nonterminal E are in the language of standard expressions, as discussed in Chapter 6. We add assignment expressions and conditionals through Rules 2, 4, and 5. Rule 3 allows a statement to resolve to an expression, whose value in this language is ignored.

Left and right values

It is a common convention in programming languages for an assignment statement $x \leftarrow y$ to have the effect of storing the *value* of variable y at *location* x : the meaning of an identifier differs across the assignment operator. On the left-hand side, x refers to the address at which x is stored; on the right-hand side, y refers to the current value stored in y . Programming languages are designed with this disparity in semantics because the prevalent case is indeed $x \leftarrow y$; consistent semantics would require special annotation on the left- or right-hand side, which would render programs difficult to read. Due to the change in meaning on the left- and right-hand sides of the assignment operator, the location and value of a variable are called the variable's **left value** (Lvalue) and **right value** (Rvalue), respectively. In Figure 7.13, the rules involving nonterminals L and R derive syntactically valid Lvalues and Rvalues:

- An assignment is generated by Rule 2, which expects an Lvalue prior to the assignment operator and an expression afterwards.
- An Lvalue is by definition any construct that can be the target of an assignment. Thus, Rule 6 defines the prevalent case of an identifier as the target of assignment.

Some languages allow addresses to be formed from values, introduced by a dereferencing operator ($*$ in C). Rule 7 can therefore also serve as an Lvalue, with the semantics of assignment to the address formed from the value R.

- The nonterminal R appears in Figure 7.13 wherever an Rvalue is expected. Semantically, an Lvalue can serve as an Rvalue by implicit dereference, as is the case for y in $x \leftarrow y$. Programming languages are usually designed to obviate the need for explicit syntax to dereference y . Transformation of an Lvalue into an Rvalue is accommodated by Rule 8. Additionally, constructs such as constants and addresses—which cannot be Lvalues—are explicitly included as Rvalues by Rules 9 and 10.

Although the dereference operator appears syntactically in Rule 7, code generation will call for dereferencing only when Rule 8 is applied. This point is emphasized in Exercise 9.

- Expressions are generated by Rule 3, with multiplication having precedence over addition. An expression (E) is a sum of one or more terms (T); a term is the product of one or more factors (F). Finally, a factor can derive either a parenthesized expression or an Rvalue. Rule 15 provides syntax to alter the default evaluation order for expressions. Rule 16 allows F to generate any construct that has an Rvalue.

7.5.2 Designing the AST

As a first attempt at an AST design, it is helpful to expose unnecessary detail by examining concrete syntax trees such as the one shown in Figure 7.14. We examine each syntactic construct and determine which portions are relevant to the meaning of a Cicada program.

Consistency

It is important that each node type in the AST have consistent meaning. Returning to our Lvalue and Rvalue discussion, an AST node representing an identifier should have consistent meaning, regardless of its occurrence in the AST. It is easy to dereference an address to obtain the current value at that address; it is not always possible to go the other way. We therefore adopt the convention that an identifier node always means the Lvalue of the identifier; if a value is desired, then the AST will require a dereference operator.

Consider the assignment $x \leftarrow y$. The AST we desire for this fragment is shown in Figure 7.15. The assignment AST node is therefore defined as a binary node

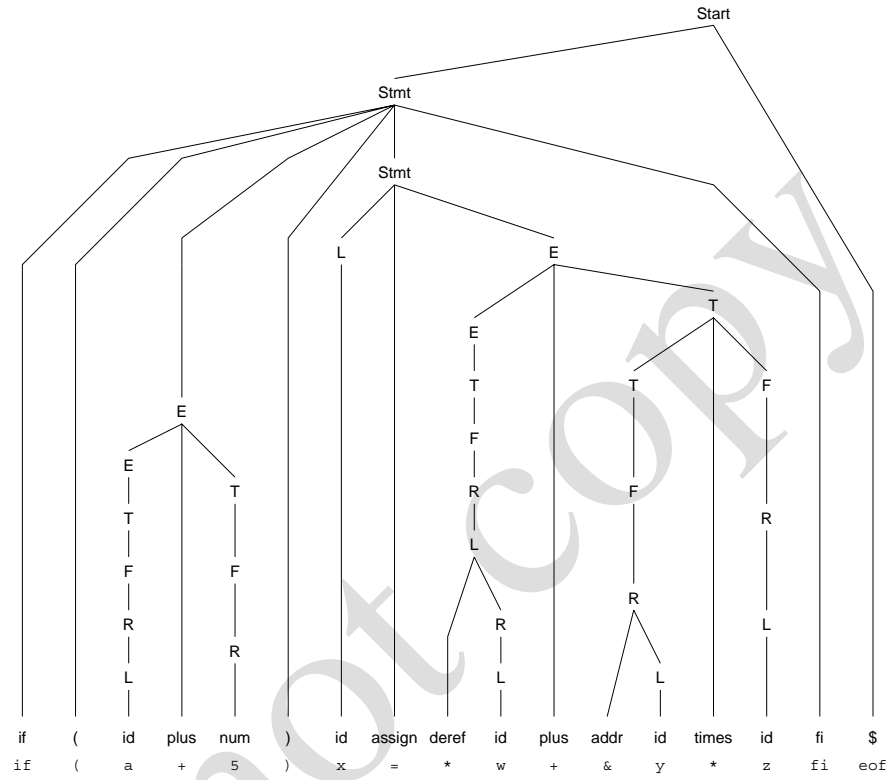


Figure 7.14: Parse tree for a Cicada program.

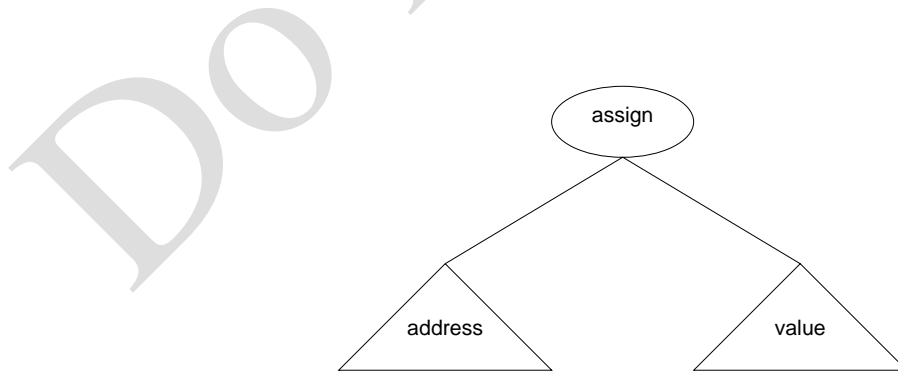


Figure 7.15: AST for a simple assignment.

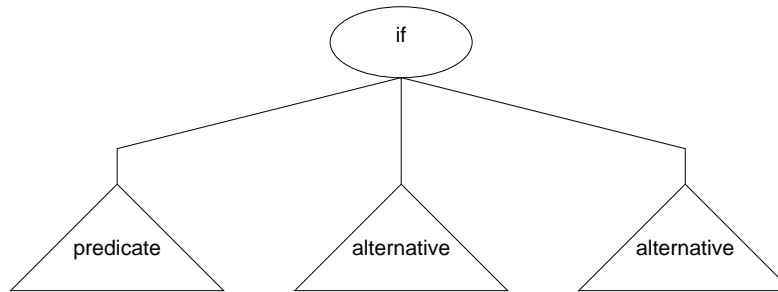


Figure 7.16: AST for a conditional statement.

(two children): its left child evaluates to the address at which the assignment should store the value obtained by evaluating the right child. In Chapter Chapter:global:codegen, code generation for such subtrees will rely upon this AST design; consistency of design will allow the same code sequence to be generated each time an identifier node is encountered.

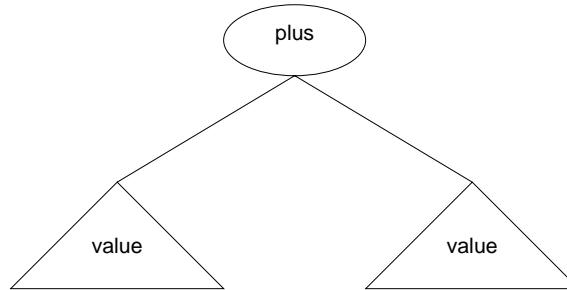
Cicada has two conditional statements, defined by Rules 4 and 5. Syntactically it is expedient to offer both forms, so that programmers need not supply an empty alternative when one is not needed. However, for consistency it may be desirable to summarize both forms of conditional statement by a single AST construct, whose alternative under a false predicate is empty.

Elimination of detail

Programming languages often contain syntactic components that serve aesthetic or technical purposes without adding any semantic value. For example, parentheses surround predicates in `if` and `while` statements in C and Java. These serve to punctuate the statement, and the closing parenthesis is needed to separate the conditional from the rest of the `if` statement; in Pascal, the keyword `then` serves this same purpose. As shown in Figure 7.16, The AST design for conditionals in Cicada retains only the predicate and the subtree for the branch executed when the predicate holds; as discussed above, a single AST construct represents the semantics of Rules 4 and 5.

Given our convention that identifiers are always interpreted as Lvalues in the AST, the address operator's appearance in an AST through Rule 10 would be redundant. Our AST design therefore does not use the address operator. Similarly, the parentheses surrounding an expression for Rule 15 are unnecessary for grouping, since the AST we create implies an evaluation order for expressions.

Unnecessary detail can occur in concrete syntax trees for the purpose of structure and disambiguation in the grammar. The rules for E, T, and F enforce the precedence of dereference over multiplication over addition; the rules also structure the parse so that arithmetic operators are left-associating. As a result, the

Figure 7.17: AST for $E \rightarrow E \text{ plus } T$.

grammar contains a number of *unit productions*: rules of the form $A \rightarrow B$ that supply no semantic content. Rules such as $E \rightarrow T$ escape the recursion on E , but the value under the E is identical to the value under the T . Moreover, the concrete subtree for $E \rightarrow E \text{ plus } T$ contains detail in excess of the sum of two values. As shown in Figure 7.17, the AST for this construct retains only the operator and the two values to be summed. The other arithmetic operators for Cicada are similarly accommodated.

7.5.3 Constructing the AST

After designing the AST, semantic actions need to be installed into the parser to construct the vertices and edges that comprise the tree. It is often useful at this point to reconsider the grammar with an eye toward simplifying AST construction. In Cicada we have decided to represent Rules 4 and 5 with the single AST construct shown in Figure 7.16. If we insert semantic actions into the grammar of Figure 7.13 we would find that Rules 4 and 5 have much of their semantic code in common. It is therefore convenient to rewrite *Stmt* as shown in Figure 7.18: a new nonterminal *CondStmt* generates the two statements previously contained directly under *Stmt*. No change to the language results from this modification; however, the semantic actions needed to construct a conditional AST node can be confined to $\text{Stmt} \rightarrow \text{CondStmt}$ if we arrange for each rule for *CondStmt* to provide a predicate and two alternatives for conditional execution.

An efficient AST data structure

While any tree data structure can in theory represent an AST, a better design can result by observing the following:

- The AST is typically constructed bottom-up. A list of siblings is generated, and the list is later adopted by a parent. The AST data structure should therefore support siblings of temporarily unknown descent.

1	Start	→	Stmt _{ast} \$ return (<i>ast</i>)	
2	Stmt _{result}	→	L _{target} assign E _{source} <i>result</i> ← MAKEFAMILY(assign, <i>source</i> , <i>target</i>)	
3			E _{expr} <i>result</i> ← <i>expr</i>	
4			CondStmt _{triple} <i>result</i> ← MAKEFAMILY(if, <i>triple.pred</i> , <i>triple.t</i> , <i>triple.f</i>)	
5	CondStmt _{triple}	→	if (E _{expr}) Stmt _{stmt} fi <i>triple.pred</i> ← <i>expr</i> <i>triple.t</i> ← <i>stmt</i> <i>triple.f</i> ← MAKECONSTNODE(0)	7
6			if (E _{expr}) Stmt _{s1} else Stmt _{s2} fi <i>triple.pred</i> ← <i>expr</i> <i>triple.t</i> ← <i>s1</i> <i>triple.f</i> ← <i>s2</i>	
7	L _{result}	→	id _{name} <i>result</i> ← MAKEIDNODE(<i>name</i>)	
8			deref R _{val} <i>result</i> ← <i>val</i>	8
9	R _{result}	→	L _{val} <i>result</i> ← MAKEFAMILY(deref, <i>val</i>)	9
10			num _{val} <i>result</i> ← MAKECONSTNODE(<i>val</i>)	
11			addr L _{val} <i>result</i> ← <i>val</i>	10
12	E _{result}	→	E _{v1} plus T _{v2} <i>result</i> ← MAKEFAMILY(plus, <i>v1</i> , <i>v2</i>)	
13			T _v <i>result</i> ← <i>v</i>	
14	T _{result}	→	T _{v1} times F _{v2} <i>result</i> ← MAKEFAMILY(times, <i>v1</i> , <i>v2</i>)	
15			F _v <i>result</i> ← <i>v</i>	
16	F _{result}	→	(E _v) <i>result</i> ← <i>v</i>	
17			R _v <i>result</i> ← <i>v</i>	

Figure 7.18: Semantic actions for AST creation in Cicada.

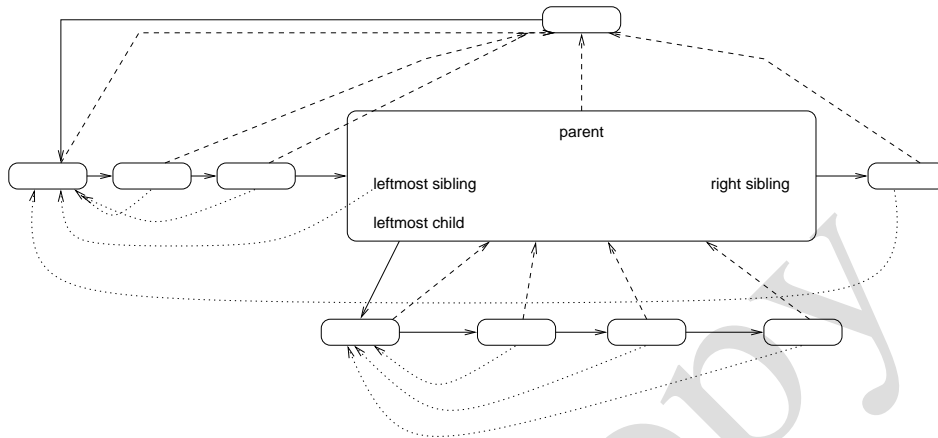


Figure 7.19: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.

- Lists of siblings are typically generated by recursive rules. The AST data structure should simplify adoption of siblings at either end of a list.
- Some AST nodes have a fixed number of children. For example, binary addition and multiplication require two children; the conditional AST node developed for Cicada requires three children. However, some programming language constructs may require an arbitrarily large number of children. Such constructs include compound statements, which accommodate zero or more statements, and method parameter and argument lists. The data structure should efficiently support tree nodes with arbitrary degree.

Figure 7.19 depicts an organization for an AST data structure that accommodates the above with a constant number of fields per node. Each node contains a reference to its parent, its next (right) sibling, its leftmost child, and its leftmost sibling.

Infrastructure for creating ASTs

To facilitate construction of an AST we rely upon the following set of constructors and methods to create and link AST nodes. create and link AST nodes. methods for creating and linking AST nodes:

`MAKECONSTNODE(val)` creates a node that represents the integer constant *val*.

`MAKEIDNODE(id)` creates a node for the identifier *id*. During symbol-table construction (Chapter Chapter:global:8), this node is often replaced by a node

that references the appropriate symbol table entry. This node refers to the Lvalue of *id*.

MAKEOPERATORNODE(*oper*) creates an AST node that represents the operation *oper*. It is expedient to specify *oper* as an actual terminal symbol in the grammar. For Cicada, *oper* could be plus, times, or deref.

x.MAKESIBLINGS(*y*) causes *y* to become *x*'s right sibling. To facilitate chaining of nodes through recursive grammar rules, this method returns a reference to the rightmost sibling resulting from the invocation.

x.HEAD() returns the leftmost sibling of *x*.

x.ADOPTCHILDREN(*y*) adopts *y* and its siblings under the parent *x*.

MAKEFAMILY(*op*, *kid*₁, *kid*₂, . . . , *kid*_{*n*}) generates a family with exactly *n* children under the operation *op*. The code for the most common case (*n* = 2) is:

```
function MAKEFAMILY(op, kid1, kid2) : node
    kids ← kid1.MAKESIBLINGS(kid2)
    parent ← MAKEOPERATORNODE(op)
    call parent.ADOPTCHILDREN(kids)
    return (parent)
end
```

The semantic actions for constructing a Cicada AST are shown in Figure 7.18. The prevalent action is to glue siblings and a parent together via MAKEFAMILY; conditionals and dereferences are processed as follows:

- The conditional statements at Rules 5 and 6 generate three values that are passed to Rule 4. These values are consistently supplied as the predicate and the two alternatives of the if statement.
- Step 7 (arbitrarily) synthesizes the constant 0 for the missing else alternative of Rule 5. The AST design should be influenced by the source language's definition, which should specify the particular value—if any—that is associated with a missing else clause.
- Step 10 executes when an object's address is taken. However, the actual work necessary to perform this task is less than one might think. Instead of expending effort to take an object's address, the *addr* operator actually *prevents* the dereference that would have been performed had the *id* been processed by the rule $R \rightarrow L$.
- Step 8 represents a syntactic dereference. However, the AST does not call for a dereference, and the subtree corresponding to the dereferenced Rvalue is simply copied as the result. While this may seem a mistake, the actual dereference occurs further down the tree, when an Lvalue becomes an Rvalue, as described next.

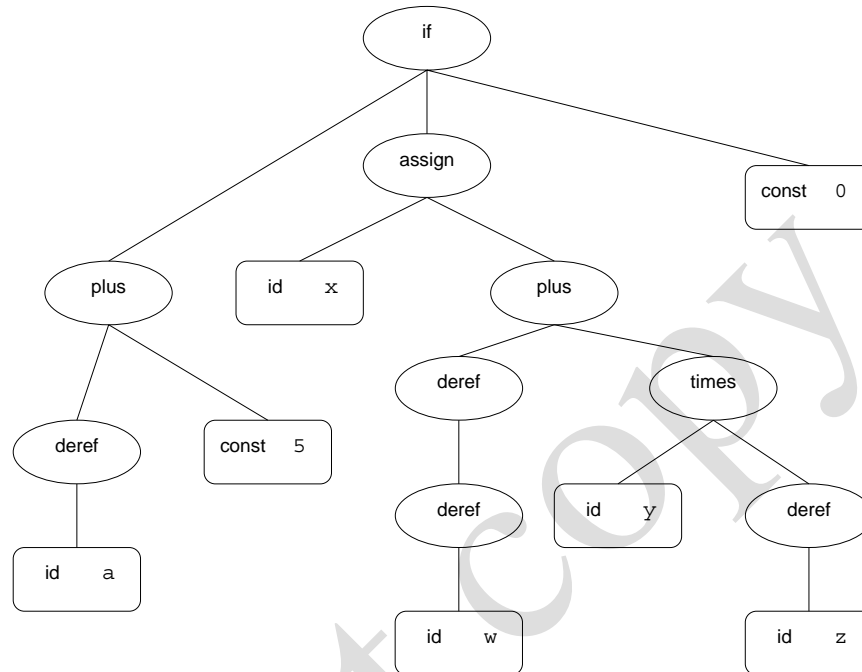


Figure 7.20: AST for the Cicada program in Figure 7.14.

- Step 9 generates an AST node that represents an actual dereference, in response to the transformation of an Lvalue into an Rvalue.
- The AST is complete at Rule 1; the root is contained in the semantic value *ast*.

Figure 7.20 shows the AST that results from the parse of Figure 7.14. The double dereference of *w* occurs in response to the input string's “* *w*”, used as an Rvalue. Indeed, the tree shown in Figure 7.14 contains two R→L productions above the terminal *w*. The use of *y* without dereference in Figure 7.20 occurs because of the *addr* operator in Figure 7.14.

List processing using left-recursive rules

A common idiom in grammars and semantic actions is the processing of a list of grammar symbols via a recursive production. In bottom-up parsing, such rules are usually left-recursive, which minimizes stack usage and creates the prevalent left-association. In top-down parsing, such rules are necessarily right-recursive.

To illustrate how such left- or right-recursive lists can be distilled into an AST we recall the grammar from Section 7.3 that defined base-10 numbers. In Figure 7.2, the semantic actions compute the value of a number. Figure 7.21 reprises

```

1  Startast  →  Digskids $
                   ast ← MAKEOPERATORNODE(Digs)
                   call ast.ADOPTCHILDREN(kids)

2  Digsup   →  Digsbelow dnext
                   sib ← MAKECONSTNODE(next)           11
                   up ← below.MAKESIBLINGS(sib)         12

3      |  dfirst
                   up ← MAKECONSTNODE(first)

```

Figure 7.21: AST construction for a left-recursive rule.

the grammar, but with the goal of generating an AST to represent the list of digits. The utility method `MAKESIBLINGS` was carefully defined to return the rightmost of a list of siblings. Thus, Step 12 returns the most recently processed digit, made into an AST node by Step 11.

Summary

Syntax-directed translation can accomplish translation directly via semantic actions that execute in concert with top-down or bottom-up parses. More commonly, an AST is constructed as a by-product of the parse; the AST serves as a record of the parse as well as a repository for information created and consumed by a compiler's passes. The design of an AST is routinely revised to simplify or facilitate compilation.

Exercises

1. Consider a right-recursive formulation for Digs of Figure 7.2, resulting in the following grammar.

$$\begin{array}{lcl}
 1 & \text{Start} & \rightarrow \text{Digs}_{ans} \$ \\
 & & \quad \text{call PRINT}(ans) \\
 2 & \text{Digs}_{up} & \rightarrow \text{d}_{next} \text{Digs}_{below} \\
 & & \quad up \leftarrow below \times 10 + next \\
 3 & & | \text{d}_{first} \\
 & & \quad up \leftarrow first
 \end{array}$$

Are the semantic actions still correct? If so, explain why they are still valid; if not, provide a set of semantic actions that does the job properly.

2. The semantic actions in Figure 7.4 fail to ensure that digits interpreted base-8 are in the range $0 \dots 7$. Show the semantic actions that need to be inserted into Figure 7.4 so that digits that lie outside the octal range are effectively ignored.
3. The grammar in Figure 7.8 is almost faithful to the language originally defined in Figure 7.6. To appreciate the difference, consider how each grammar treats the input string $x \ 5 \ \$$.
 - (a) In what respects do the grammars generate different languages?
 - (b) Modify the grammar in Figure 7.8 to maintain its style of semantic processing but to respect the language definition in Figure 7.6.
4. The language generated by the grammar in Figure 7.8 uses the terminal x to introduce the base. A more common convention is to separate the base from the string of digits by some terminal symbol. Instead of $x \ 8 \ 4 \ 3 \ 1$ to represent 431_8 , a language following the common convention would use $8 \ x \ 4 \ 3 \ 1$.
 - (a) Design an LALR(1) grammar for such a language and specify the semantic actions that compute the string's numeric value. In your solution, allow the absence of a base specification to default to base 10, as in Figure 7.8.
 - (b) Discuss the tradeoffs of this language and your grammar as compared with the language and grammar of Figure 7.8.
5. Consider the addition of the the rule

$$\text{Expr} \rightarrow \text{sum Values}$$

to the grammar in Figure 7.9.

- (a) Does this change make the grammar ambiguous?
- (b) Is the grammar still LL(1) parsable?
- (c) Show how the semantic actions in Figure 7.10 must be changed to accommodate this new language construct; modify the grammar if necessary but avoid use of global variables.

6. Consider the addition of the rule

$$\text{Expr} \rightarrow \text{mean Values}$$

to the grammar in Figure 7.9. This rule defines an expression that computes the average of its values, defined by:

$$(\text{mean } v_1 v_2 \dots v_n) = \left[\frac{\sum_{i=1}^n v_i}{n} \right]$$

- (a) Does this render the grammar ambiguous?
 - (b) Is the grammar still LL(1) parsable?
 - (c) Describe your approach for syntax-directed translation of this new construct.
 - (d) Modify the grammar of Figure 7.10 accordingly and insert the appropriate semantic actions into Figure 7.10.
7. Although arithmetic expressions are typically evaluated from left to right, the semantic actions in VALUES cause a product computed from right to left. Modify the grammar and semantic actions of Figures 7.9 and 7.10 so that products are computed from left to right.
8. Verify that the grammar in Figure 7.13 is unambiguous using an LALR(1) parser generator.
9. Suppose that the terminals `assign`, `deref`, and `addr` correspond to the input symbols `=`, `*`, and `&`, respectively. Using the grammar in Figure 7.13
- show parse trees for the following strings;
 - show where indirections actually occur by circling the parse tree nodes that correspond to the rule $R \rightarrow L$.
- (a) `x = y`
 - (b) `x = * y`
 - (c) `* x = y`
 - (d) `* x = * y`
 - (e) `* * x = & y`
 - (f) `* 16 = 256`

10. Construct an LL(1) grammar for the language in Figure 7.13.
11. Suppose Cicada were extended to include binary subtraction and unary negation, so that the expression `minus y minus x times 3` has the effect of negating `y` prior to subtraction of the product of `x` and 3.
 - (a) Following the steps outlined in Section 7.5, modify the grammar to include these new operators. Your grammar should grant negation strongest precedence, at a level equivalent to `deref`. Binary subtraction can appear at the same level as binary addition.
 - (b) Modify the chapter's AST design to include subtraction and negation by including a minus operator node.
 - (c) Install the appropriate semantic actions into the parser.
12. Modify the grammar and semantic actions of Figure 7.21 so that the rule for `Digs` is right-recursive. Your semantic actions should create the identical AST.
13. Using a standard LALR(1) grammar for C or Java, find syntactic punctuation that can be eliminated without introducing LALR(1) conflicts. Explain why the (apparently unnecessary) punctuation is included in the language. As a hint, consider the parentheses that surround the predicate of an `if` statement.

8

Symbol Tables

Chapter Chapter:global:seven considered the construction of an **abstract syntax tree** (AST) as an artifact of a top-down or bottom-up parse. On its own, a top-down or bottom-up parser cannot fully accomplish the compilation of modern programming languages. The AST serves to represent the source program and to coordinate information contributed by the various passes of a compiler. This chapter presents one such pass—the harvesting of **symbols** from an AST. Most programming languages allow the declaration, definition, and use of symbolic names to represent constants, variables, methods, types, and objects. The compiler checks that such names are used correctly, based on the programming language's definition. This chapter describes the organization and implementation of a **symbol table**. This structure records the names and important attributes of a program's names. Examples of such attributes include a name's type, scope, and accessibility.

We are interested in two aspects of a symbol table: its use and its organization. Section 8.1 defines a simple symbol table interface and shows how to use this interface to manage symbols for a block-structured language. Section 8.2 explains the effects of program *scopes* on symbol-table management. Section 8.3 examines various implementations of a symbol table. Advanced topics, such as type definitions, inheritance, and overloading are considered in Section 8.4.

8.1 Use of a Symbol Table

In this section, we consider how to construct a symbol table for a simple, block-structured language. Assuming an AST has been constructed as described in Chapter Chapter:global:seven, we **walk** (make a **pass** over) the AST to

- process symbol declarations and

- connect each symbol reference with its declaration.

Symbol references are connected with declarations through the symbol table. An AST node that mentions a symbol by name is enriched with a reference to the name's entry in the symbol table. If such a connection cannot be made, then the offending reference is improperly declared and an error message is issued. Otherwise, subsequent passes can use the symbol table reference to obtain information about the symbol, such as its type, storage requirements, and accessibility.

The block-structured program shown in Figure 8.1(a) contains two nested scopes. Although the program uses keywords such as `float` and `int`, no symbol table action is required for these symbols if they are recognized by the scanner as the terminal symbols `float` and `int`, respectively. Most programming-language grammars demand such precision of the scanner to avoid ambiguity in the grammar.

The program in Figure 8.1(a) begins by *importing* function definitions for `f` and `g`. The compiler finds these, determines that their return types are `void`, and then records the functions in the symbol table as its first two entries. The declarations for `w` and `x` in the outer scope are entered as symbols 3 and 4, respectively. The inner scope's redeclaration of `x` and its declaration of `z` are entered as symbols 5 and 6, respectively. The AST in Figure 8.1(b) refers to symbols by name. In Figure 8.1(d), the names are replaced by symbol table references. In particular, the references to `x`—shown only by name in Figure 8.1(b)—are declaration-specific in Figure 8.1(d). In the symbol table, the references contain the original name as well as type information processed from the symbol declarations.

8.1.1 Static Scoping

Modern programming languages offer **scopes** to confine the activity of a name to a prescribed region of a program. A name may be declared at most once in any given scope. For statically scoped, block-structured languages, references are typically resolved to the declaration in their closest containing scope. Additionally, most languages contain directives to promote a given declaration or reference to the program's **global scope**—a name space shared by all compilation units. Section 8.4.4 discusses these issues in greater detail.

Proper use of scopes results in programs whose behavior is easier to understand. Figure 8.1(a) shows a program with two nested scopes. Declared in the program's outer scope, `w` is available in both scopes. The activity of `z` is confined to the inner scope. The name `x` is available in both scopes, but the *meaning* of `x` changes—the inner scope redeclares `x`. In general, the **static scope** of an identifier includes its defining block as well as any contained blocks that do not themselves contain a declaration for the identifier.

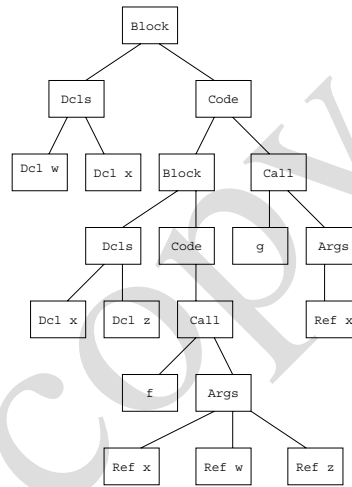
Languages such as C, Pascal, and Java use the following keywords or punctuation to define static scopes.

- For Pascal, the reserved keywords `begin` and `end` open and close scopes, respectively. Within each scope, types, variables, and methods can be declared.

```

import f(float, float, float)
import g(int)
{
  int w,x
  {
    float x,z
    f(x,w,z)
  }
  g(x)
}
    
```

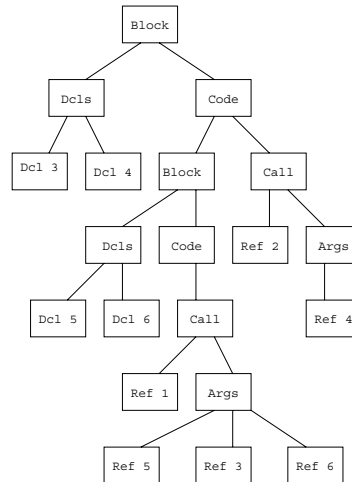
(a)



(b)

Symbol Number	Symbol Name	Attributes
1	f	void func(float, float, float)
2	g	void func(int)
3	w	int
4	x	int
5	x	float
6	z	float

(c)



(d)

Figure 8.1: Symbol table processing for a block-structured program.

- For C and Java, scopes are opened and closed by the appropriate braces, as shown in Figure 8.1(a). In these languages, a scope can declare types and variables. However, methods (function definitions) cannot appear in inner scopes.

As considered in Exercises 1 and 2, compilers sometimes create scopes to make room for temporaries and other compiler-generated names. For example, the contents of a register may be temporarily deposited in a compiler-generated name to free the register for some other task. When the task is finished, the register is reloaded from the temporary storage. A scope nicely delimits the useful life of such temporaries.

In some languages, references to names in outer scopes can incur overhead at runtime. As discussed in Chapter Chapter:global:eleven, an important consideration is whether a language allows the *nested declaration* of methods. C and Java prohibit this, while Pascal allows methods to be defined in any scope. For C and Java, a method's local variables can be **flattened** by renaming nested symbols and moving their declaration to the method's outermost scope. Exercise 3 considers this in greater detail.

8.1.2 A Symbol Table Interface

A symbol table is responsible for tracking *which* declaration is in effect when a reference to the symbol is encountered. In this section, we define a symbol-table interface for processing symbols in a block-structured, statically scoped language. The methods in our interface are as follows.

`INCRNESTLEVEL()` opens a new scope in the symbol table. New symbols are entered in the resulting scope.

`DECRNESTLEVEL()` closes the most recently opened scope in the symbol table. Symbol references subsequently revert to outer scopes.

`ENTERSYMBOL(name, type)` enters the *name* in the symbol table's current scope. The parameter *type* conveys the data type and access attributes of *name*'s declaration.

`RETRIEVESYMBOL(name)` returns the symbol table's currently valid declaration for *name*. If no declaration for *name* is currently in effect, then a *null* pointer can be returned.

To illustrate the use of this interface, Figure 8.2 contains code to build the symbol table for the AST shown in Figure 8.1. The code is specialized to the type of the AST node. Actions can be performed both before and after a given node's children are visited. Prior to visiting the children of a Block node, Step 1 increases the static scope depth. After the subtree of the Block is processed, Step 2 abandons the scope. The code for Ref retrieves the symbol's current definition in the symbol table. If none exists, then an error message is issued.

```

procedure BUILDSYMBOLTABLE( )
  call PROCESSNODE(ASTroot)
end
procedure PROCESSNODE(node)
  switch (node.kind)
    case Block
      call symtab.INCRNESTLEVEL( )           1
    case Decl
      call symtab.ENTERSYMBOL(name, type)
    case Ref
      sym ← symtab.RETRIEVESYMBOL(name)
      if sym = ⊥
        then call ERROR(“Undeclared symbol”)
      foreach c ∈ Children(node) do call PROCESSNODE(c)
      switch (node.kind)
        case Block
          call symtab.DECRNESTLEVEL( )       2
        case Decl
          call symtab.ENTERSYMBOL(name, type)
        case Ref
          call symtab.RETRIEVESYMBOL(name)
      end
    end
  end

```

Figure 8.2: Building the symbol table.

8.2 Block-Structured Languages and Scope Management

Most programming languages allow scopes to be nested statically, based on concepts introduced by Algol 60. Languages that allow nested name scopes are known as **block-structured languages**. While the mechanisms that open and close scopes can vary by language, we assume that the `INCRNESTLEVEL` and `DECRNESTLEVEL` methods are the uniform mechanism for opening and closing scopes in the symbol table. In this section, we consider various language constructs that call for the opening and closing of scopes. We also consider the issue of allocating a symbol table per scope as compared with a single, global symbol table.

8.2.1 Scopes

Every symbol reference in an AST occurs in the context of defined scopes. The scope defined by the *innermost* such context is known as the **current scope**. The scopes defined by the current scope and its surrounding scopes are known as the **open** or **currently active scopes**. All other scopes are said to be **closed**. Based on these definitions, current, open, and closed scopes are not fixed attributes; instead, they are defined relative to a particular point in the program.

Following are some common **visibility rules** that define the interpretation of a name in the presence of multiple scopes.

- At any point in the text of a program, the accessible names are those that are declared in the current scope and in all other open scopes.
- If a name is declared in more than one open scope, a reference to the name is resolved to the *innermost* declaration—the one that most closely surrounds the reference.
- New declarations can be made only in the current scope.

Most languages offer mechanisms to install or resolve symbol names in the outermost, program-global scope. In C, names bearing the `extern` attribute are resolved globally. In Java, a class can reference any class's `public static` fields, but these fields do not populate a single, flat name space. Instead, each such field must be fully qualified by its containing class.

Programming languages have evolved to allow various useful levels of scoping. C and C++ offer a compilation-unit scope—names declared outside of all methods are available within the compilation unit's methods. Java offers a package-level scope—classes can be organized into packages that can access the package-scoped methods and fields. In C, every function definition is available in the global scope, unless the definition has the `static` attribute. In C++ and Java, names declared within a class are available to all methods in the class. In Java and C++, a class's fields and methods bearing the `protected` attribute are available to the class's subclasses. The parameters and local variables of a method are available within the given method. Finally, names declared within a statement-block are available in all contained blocks, unless the name is redeclared in an inner scope.

8.2.2 One Symbol Table or Many?

There are two common approaches to implementing block-structured symbol tables, as follows.

- A symbol table is associated with each scope.
- All symbols are entered in a single, global table.

A single symbol table must accommodate multiple, active declarations of the same symbol. On the other hand, searching for a symbol can be faster in a single symbol table. We next consider this issue in greater detail.

An Individual Table for Each Scope

If an individual table is created for each scope, some mechanism must be in place to ensure that a search produces the name defined by the nested-scope rules. Because name scopes are opened and closed in a **last-in, first-out** (LIFO) manner, a stack is an appropriate data structure for organizing such a search. Thus, a **scope stack** of symbol tables can be maintained, with one entry in the stack for each open scope. The innermost scope appears at the top of the stack. The next containing scope is

second from the top, and so forth. When a new scope is opened, `INCRNESTLEVEL` creates and pushes a new symbol table on the stack. When a scope is closed, `DECRNESTLEVEL`, the top symbol table is popped.

A disadvantage of this approach is that we may need to search for a name in a number of symbol tables before the symbol is found. The cost of this stack search varies from program to program, depending on the number of nonlocal name references and the depth of nesting of open scopes. In fact, studies have shown that most lookups of symbols in block-structured languages return symbols in the inner- or outer-most scopes. With a table per scope, intermediate scopes must be checked before an outermost declaration can be returned.

An example of this symbol table organization is shown in Figure 8.8.

One Symbol Table

In this organization, all names in a compilation unit's scopes are entered into the same table. If a name is declared in different scopes, then the scope name or depth helps identify the name uniquely in the table. With a single symbol table, `RETRIEVESYMBOL` need not chain through scope tables to locate a name. Section 8.3.3 describes this kind of symbol table in greater detail. Such a symbol table is shown in Figure 8.7.

8.3 Basic Implementation Techniques

Any implementation of the interface presented in Section 8.1 must correctly insert and find symbols. Depending on the number of names that must be accommodated and other performance considerations, a variety of implementations is possible. Section 8.3.1 examines some common approaches for organizing symbols in a symbol table. Section 8.3.2 considers how to represent the symbol names themselves. Based on this discussion, Section 8.3.3 proposes an efficient symbol table implementation.

8.3.1 Entering and Finding Names

We begin by considering various approaches for *organizing* the symbols in the symbol table. For each approach, we examine the time needed to

- insert symbols,
- retrieve symbols, and
- maintain scopes.

These actions are not typically performed with equal frequency. Each scope can declare a name at most once, but names can be referenced multiple times. It is therefore reasonable to expect that `RETRIEVESYMBOL` is called more frequently than the other methods in our symbol table interface. Thus, we pay particular attention to the cost of retrieving symbols.

Unordered List

This is the simplest possible storage mechanism. The only data structure required is an array, with insertions occurring at the next available location. For added flexibility, a linked list avoids the limitations imposed by a fixed-size array. In this representation, ENTERSYMBOL inserts a name at the head of the unordered list. The scope name (or depth) is recorded with the name. This allows ENTERSYMBOL to detect if the same name is entered twice in the same scope—a situation disallowed by most programming languages. RETRIEVESYMBOL searches for a name from the head of the list toward its tail, so that the closest, active declaration of the name is encountered first.

All names for a given scope appear adjacently in the unordered list. Thus, INCRNESTLEVEL can annotate the list with a marker to show where the new scope begins. DECRNESTLEVEL can then delete the currently active symbols at the head of the list.

Although insertion is fast, retrieval of a name from the outermost scope can require scanning the entire unordered list. This approach is therefore impractically slow except for the smallest of symbol tables.

Ordered List

If a list of n distinct names is maintained alphabetically, binary search can find any name in $O(\log n)$ time. In the unordered list, declarations from the same scope appear in sequence—an unlikely situation for the ordered list. How should we organize the ordered list to accommodate declarations of a name in multiple scopes? Exercise 4 investigates the potential performance of storing all names in a single, ordered list. Because RETRIEVESYMBOL accesses the *currently active* declaration for a name, a better data structure is an ordered list of *stacks*. Each stack represents one currently active name; the stacks are ordered by their representative names.

RETRIEVESYMBOL locates the appropriate stack using binary search. The currently active declaration appears on top of the located stack. DECRNESTLEVEL must pop those stacks containing declarations for the abandoned scope. To facilitate this, each symbol can be recorded along with its scope name or depth, as established by INCRNESTLEVEL. DECRNESTLEVEL can then examine each stack in the list and pop those stacks whose top symbol is declared in the abandoned scope. When a stack becomes empty, it can be removed from the ordered list. Figure 8.3 shows such a symbol table for the example in Figure 8.1, at the point where method `f` is invoked.

A more efficient approach avoids touching each stack when a scope is abandoned. The idea is to maintain a separate linking of symbol table entries that are declared at the same scope level. Section 8.3.3 presents this organization in greater detail.

The details of maintaining a symbol table using ordered lists is left as Exercise 5. Although ordered lists offer fast retrieval, insertion into an ordered list

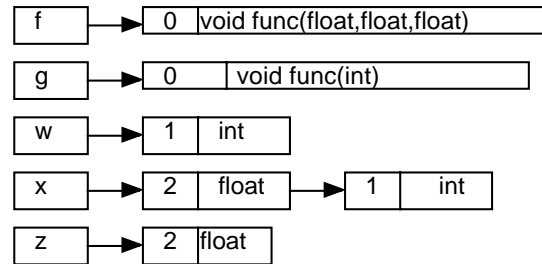


Figure 8.3: An ordered list of symbol stacks.

is relatively expensive. Thus, ordered lists are advantageous when the space of symbols is known in advance—such is the case for reserved keywords.

Binary Search Trees

Binary search trees are designed to combine the efficiency of a linked data structure for insertion with the efficiency of binary search for retrieval. Given random inputs, it is expected that a name can be inserted or found in $O(\log n)$ time, where n is the number of names in the tree. Unfortunately, average-case performance does not necessarily hold for symbol tables—programmers do not choose identifier names at random! Thus, a tree of n names could have depth n , causing name lookup to take $O(n)$ time.

An advantage of binary search trees is their simple, widely known implementation. This simplicity and the common perception of reasonable average-case performance make binary search trees a popular technique for implementing symbol tables. As with the list structures, each name (node) in the binary search tree is actually a *stack* of currently active scopes that declare the name.

Balanced Trees

The worst-case scenario for a binary search tree can be avoided if a search tree can be maintained in *balanced* form. The time spent balancing the tree can be amortized over all operations so that a symbol can be inserted or found in $O(\log n)$ time, where n is the number of names in the tree. Examples of such trees include **red-black trees** [?] and **splay trees** [?]. Exercises 10 and 11 further explore symbol table implementations based on balanced-tree structures.

Hash Tables

Hash tables are the most common mechanism for managing symbol tables, owing to their excellent performance. Given a sufficiently large table, a good hash function, and appropriate collision-handling techniques, insertion or retrieval can be

performed in *constant* time, regardless of the number of entries in the table. The implementation discussed in Section 8.3.3 uses a hash table, with collisions handled by chaining. Hash tables are widely implemented. Some languages (including Java) contain hash table implementations in their core library. The implementation details for hash tables are covered well in most books on elementary data structures and algorithms [?].

8.3.2 The Name Space

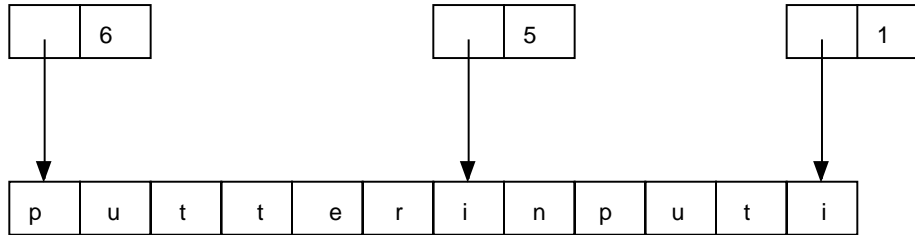
At some point, a symbol table entry must represent the *name* of its symbol. Each name is essentially a string of characters. However, by taking the following properties into consideration, an efficient implementation of the name space can be obtained.

- The name of a symbol does not change during compilation. Thus, the strings associated with symbol table names are **immutable**—once allocated, they do not change.
- Although scopes come and go, the symbol names persist throughout compilation. Scope creation and deletion affects the set of currently available symbols, obtainable through `RETRIEVESYMBOL`. However, a scope's symbols are not completely forgotten when the scope is abandoned. Space must be reserved for the symbols at runtime, and the symbols may require initialization. Thus, the symbols' strings occupy storage that persists throughout compilation.
- There can be great variance in the *length* of identifier names. Short names—perhaps only a single character—are typically used for iteration variables and temporaries. Global names in a large software system tend to be descriptive and much longer in length. For example, the X windowing system contains names such as `VisibilityPartiallyObscured`.
- Unless an ordered list is maintained, comparisons of symbol names involve only equality and inequality.

The above points argue in favor of one logical name space, as shown in Figure 8.4, in which names are inserted but never deleted.

In Figure 8.4, each string referenced by a pair of fields. One field specifies the string's origin in the string buffer, and the other field specifies the string's length. If the names are managed so that the buffer contains at most one occurrence of any name, then the equality of two strings can be tested by comparing the strings' references. If they differ in origin or length, the strings cannot be the same. The Java `String` class contains the method `intern` that maps any string to a unique reference for the string.

The strings in Figure 8.4 do not share any common characters. Exercise 16 considers a `stringspace` that stores shared substrings more compactly. In some languages, the suffix of a name can suffice to locate the name. For example, a

Figure 8.4: Name space for symbols `putter`, `input`, and `i`.

Name	Type	Var	Level	Hash

Figure 8.5: A symbol table entry.

reference of `String` in a Java program defaults to `java.lang.String`. Exercise 6 considers the organization of name spaces to accommodate such access.

8.3.3 An Efficient Symbol Table Implementation

We have examined issues of symbol management and representation. Based on the discussion up to this point, we next present an efficient symbol table implementation. Figure 8.5 shows the layout of a symbol table entry, each containing the following fields.

Name is a reference to the symbol name space, organized as described in Section 8.3.2. The name is required to locate the symbol in a chain of symbols with the same hash-table location.

Type is a reference to the type information associated with the symbol's declaration. Such information is processed as described in Chapter Chapter:global:nine.

Hash threads symbols whose names hash to the same value. In practice, such symbols are doubly linked to facilitate symbol deletion.

Var is a reference to the next outer declaration of this same name. When the scope containing this declaration is abandoned, the referenced declaration becomes the currently active declaration for the name. Thus, this field essentially represents a stack of scope declarations for its symbol name.

Level threads symbols declared in the same scope. This field facilitates symbol deletion when a scope is abandoned.

There are two *index* structures for the symbol table: a hash table and a scope display. The hash table allows efficient lookup and entry of names, as described in Section 8.3.1. The **scope display** maintains a list of symbols that are declared at the same level. In particular, the *i*th entry of the scope display references a list of symbols currently active at scope depth *i*. Such symbols are linked by their Level field. Moreover, each active symbol's Var field is essentially a stack of declarations for the associated variable.

Figure 8.6 shows the pseudocode for this symbol table implementation. Figure 8.7 shows the table that results from applying this implementation to the example in Figure 8.1, at the point where method *f* is invoked. Figure 8.7 assumes the following unlikely situation with respect to the hash function.

- *f* and *g* hash to the same location.
- *w* and *z* hash to the same location.
- All symbols are clustered in the same part of the table.

The code in Figure 8.6 relies on the following methods.

DELETE(*sym*) removes the symbol table entry *sym* from the collision chain found at *HashTable.GET(sym.name)*. The symbol is not destroyed—it is simply removed from the collision chain. In particular, its Var and Level fields remains intact.

ADD(*sym*) adds the symbol entry *sym* to the collision chain at *HashTable.GET(sym.name)*. Prior to the call to ADD, there is no entry in the table for *sym*.

When **DECRNESTLEVEL** is invoked to abandon the currently active scope, each symbol in the scope is visited by the loop at Step 3. Each such symbol is removed from the hash table at Step 4. If an outer scope definition exists for the symbol, the definition is inserted into the hash table at Step 5. Thus, the Var field serves to maintain a stack of active scope declarations for each symbol. The Level field allows **DECRNESTLEVEL** to operate in time proportional to the number of symbols affected by abandoning the current scope. Amortized over all symbols, this adds a constant overhead to the management of each declared symbol.

RETRIEVESYMBOL examines a collision chain to find the desired symbol. The loop at Step 6 accesses all symbols that hash to the same table location—the chain that should contain the desired *name*. Step 7 follows the entries' Hash fields until the chain is exhausted or the symbol is located. A properly managed hash table should have very short collision chains. Thus, we expect that only a few iterations of the loop at Step 6 should be necessary to locate a symbol or to detect that the symbol has not been properly declared.

ENTERSYMBOL first locates the currently active definition for *name*, should any exist in the table. Step 9 checks that no declaration already exists in the current

scope. A new symbol table entry is generated at Step 10. The symbol is added to those in the current scope by linking it into the scope display. The remaining code inserts the new symbol into the table. If an active scope contains a definition of the symbol name, that name is removed from the table and referenced by the Var field of the new symbol.

Recalling the discussion of Section 8.2.2, an alternative approach segregates symbols by scope. A stack of symbol tables results—one per scope—as shown in Figure 8.8. The code to manage such a structure is left as Exercise 19.

8.4 Advanced Features

We next examine how to extend the simple symbol table framework to accommodate advanced features of modern programming languages. Extensions to our simple framework fall generally in the following categories:

- Name augmentation (overloading)
- Name hiding and promotion
- Modification of search rules

In each case, it is appropriate to rethink the symbol table design to arrive at an efficient, correct implementation of the desired features. In the following sections, we discuss the essential issues associated with each feature. However, we leave the details of the implementation as exercises.

8.4.1 Records and Typenames

Languages such as C and Pascal allow aggregate data structures using the `struct` and `record` type constructors. Because such structures can be nested, access to a field may involve navigating through many containers before the field can be reached. In Pascal, Ada, and C, such fields are accessed by completely specifying the containers and the field. Thus, the reference `a.b.c.d` accesses field `b` of record `a`, field `c` of record `b`, and finally field `d` of record `c`. COBOL and PL/I allow intermediate containers to be omitted—if the reference can be unambiguously resolved. In such languages, `a.b.c.d` might be abbreviated as `a.d` or `c.d`. This idea has not met with general acceptance, partly because programs that use such abbreviations are difficult to read. It is also possible that `a.d` is a *mistake*, but the compiler silently interprets the reference by filling in missing containers.

Records can be nested arbitrarily deep. Thus, records are typically implemented using a tree. Each record is represented as a node; its children represent the record's subfield. Alternatively, a record can be represented by a symbol table whose entries are the record's subfields. Exercise 14 considers the implementation of records in symbol tables.

C offers the `typedef` construct, which establishes a name as an *alias* for a type. As with record types, it is convenient to establish an entry in the symbol table for

```

procedure INCRNESTLEVEL( )
    Depth  $\leftarrow$  Depth + 1
    ScopeDisplay[Depth]  $\leftarrow$   $\perp$ 
end
procedure DECRNESTLEVEL( )
    foreach sym  $\in$  ScopeDisplay[Depth] do           3
        prevsym  $\leftarrow$  sym.var
        call DELETE(sym)                               4
        if prevsym  $\neq$   $\perp$                                5
            then call ADD(prevsym)
        Depth  $\leftarrow$  Depth - 1
    end
function RETRIEVESYMBOL(name) : Symbol
    sym  $\leftarrow$  HashTable.GET(name)
    while sym  $\neq$   $\perp$  do                               6
        if sym.name = name
            then return (sym)
        sym  $\leftarrow$  sym.Hash                             7
    return ( $\perp$ )                                       8
end
procedure ENTERSYMBOL(name, type)
    oldsym  $\leftarrow$  RETRIEVESYMBOL(name)
    if oldsym  $\neq$   $\perp$  and oldsym.depth = Depth       9
        then call ERROR(“Duplicate declaration of name”)
    newsym  $\leftarrow$  CREATENEWSYMBOL(name, type)       10
    /*          Add to scope display          */
    newsym.Level  $\leftarrow$  ScopeDisplay[Depth]
    ScopeDisplay[Depth]  $\leftarrow$  newsym
    /*          Add to hash table          */
    if oldsym =  $\perp$ 
        then call ADD(newsym)
    else
        call DELETE(oldsym)
        call ADD(newsym)
    newsym.var  $\leftarrow$  oldsym
end

```

Figure 8.6: Symbol table management.

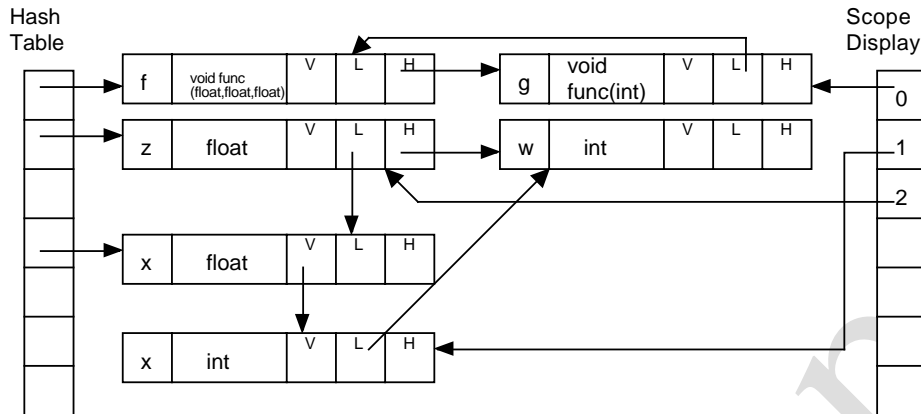


Figure 8.7: Detailed layout of the symbol table for Figure 8.1. The V, L, and H fields abbreviate the Var, Level, and Hash fields, respectively.

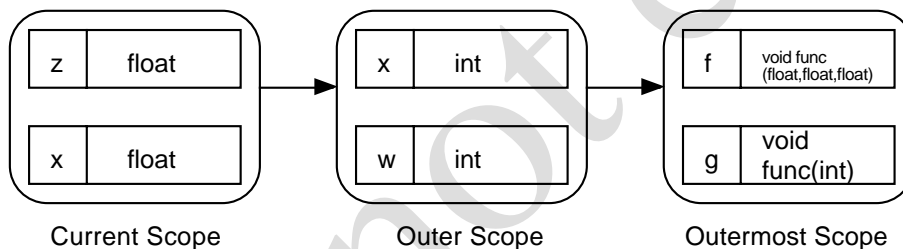


Figure 8.8: A stack of symbol tables, one per scope.

the `typedef`. In fact, most C compilers use scanners that must distinguish between ordinary names and typename. This is typically accomplished through a “back door” call to the symbol table to lookup each identifier. If the symbol table shows that the name as an active typename, then the scanner returns a typename token. Otherwise, an ordinary identifier token is returned.

8.4.2 Overloading and Type Hierarchies

The notion of an *identifier* has thus far been restricted to a string containing the identifier's name. Situations can arise where the name alone is insufficient to locate a desired symbol. Object-oriented languages such as C++ and Java allow **method overloading**. A method can be defined multiple times, provided that each definition has a unique *type signature*. The **type signature** of a method includes the number

and types of its parameters and its return type. With overloading, a program can contain the methods `print(int)` as well as `print(String)`.

When the type signatures are included, the compiler comes to view a method definition not only in terms of its name but also in terms of its type signature. The symbol table must be capable of entering and retrieving the appropriate symbol for a method. In one approach to method overloading, the type signature of a method is encoded along with its name. For example, the method `println` that accepts an integer and returns nothing is encoded as `println(I)V`. It then becomes necessary to include a method's type signature each time such symbols are retrieved from the symbol table. Alternatively, the symbol table could simply record a method along with a list of its overloaded definitions. In the AST, method invocations point to the entire list of method definitions. Subsequently, semantic processing scans the list to make certain that a valid definition of the method occurs for each invocation.

Some languages, such as C++ and Ada, allow operator symbols to be overloaded. For example, the meaning of `+` could change if its arguments are matrices instead of scalars. A program could provide a method that overloads `+` and performs matrix addition on matrix arguments. The symbol table for such languages must be capable of determining the definition of an operator symbol in every scope.

Ada allows literals to be overloaded. For example, the symbol `Diamonds` could participate simultaneously in two different enumeration types: as a playing card and as a commodity.

Pascal and FORTRAN employ a small degree of overloading in that the same symbol can represent the invocation of a method as well as the value of the method's result. For such languages, the symbol table contains two entries. One represents the method while the other represents a name for the value returned by the method. It is clear in context whether the name means the value returned by the method or the method itself. As demonstrated in Figure 8.1, semantic processing makes an explicit connection between a name and its symbol.

C also has overloading to a certain degree. A program can use the same name as a local variable, a `struct` name, and a label. Although it is unwise to write such confusing programs, C allows this because it is clear in each context which definition of a name is intended (see Exercise 13).

Languages such as Java and C++ offer type extension through subclassing. The symbol table could contain a method `resize(Shape)`, while the program invokes the method `resize(Rectangle)`. If `Rectangle` is a subclass of `Shape`, then the invocation should resolve to the method `resize(Shape)`. However, if the program contains a `resize` method for both `Rectangle` and `Shape` types, then resolution should choose the method whose formal parameters most closely match the types of the supplied parameters.

In modern, object-oriented languages, a method call's resolution is determined not only by its type signature, but also by its situation in the type hierarchy. For example, consider a Java class `A` and its subclass `B`. Suppose classes `A` and `B` both define the method `sqrt(int)`. Given an instance `b` of class `B`, an invocation of `((A) b).sqrt()` causes `B`'s `sqrt` to execute, even though the instance is apparently

converted to an object of type A. This is because the *runtime* type identification of the instance determines which class's `sqrt` is executed. At compile-time it is generally undecidable which version of `sqrt` will execute at runtime. However, the compiler can check that a valid definition of `sqrt` is present for this example. The instance of type B is regarded as an instance of class A after the cast. The compiler can check that class A provides a suitable definition of `sqrt`. If so, then all of A's subclasses provide a default definition by inheritance, and perhaps a more specialized definition by overriding the method. In fact, Java provides no mechanism for directly invoking A's definition of `sqrt` on an object whose runtime type is B. In contrast, C++ allows a *nonvirtual* call to A's definition of `sqrt`. In fact, such is the default for a B object once it is cast to type A. Chapter Chapter:global:thirteen considers these issues in greater detail.

8.4.3 Implicit Declarations

In some languages, the appearance of a name in a certain context serves to declare the name as well. As a common example, consider the use of *labels* in C. A label is introduced as an identifier followed by a colon. Unlike Pascal, the program need not declare the use of such labels in advance. In FORTRAN, the type of an identifier for which no declaration is offered can be inferred from the identifier's first letter. In Ada, a `for` loop index is implicitly declared to be of the same type as the range specifier. Moreover, a *new scope* is opened for the loop so that the loop index cannot clash with an existing variable (see Exercise 12).

Implicit declarations are almost always introduced in a programming language for convenience—the convenience of those who *use* rather than *implement* the language. Taking this point further, implicit declarations may ease the task of *writing* programs at the expense of those who must later read them. In any event, the compiler is responsible for supporting such features.

8.4.4 Export and Import Directives

Export rules allow a programmer to specify that some local scope names are to become visible outside that scope. This selective visibility is in contrast to the usual block-structured scope rule, which causes local scope names to be *invisible* outside the scope. Export rules are typically associated with modularization features such as Ada packages, C++ classes, C compilation units, and Java classes. These language features help a programmer organize a program's files by their functionality.

In Java, the `public` attribute causes the associated field or method to be known outside its class. To prevent name clashes, each class can situate itself in a package hierarchy through the `package` directive. Thus, the `String` class provided in the Java core library is actually part of the `java.lang` package. In contrast, all methods in C are known outside of their compilation units, *unless* the `static` attribute is bestowed. The `static` methods are available only within their compilation units.

With an export rule, each compilation unit advertises its offerings. In a large software system, the space of available global names can become polluted and

disorganized. To manage this, compilation units are typically required to specify which names they wish to *import*. In C and C++, the use of a header file includes declarations of methods and structures that can be used by the compilation unit. In Java, the `import` directive specifies the classes and packages that a compilation unit might access. Ada's `use` directive serves essentially the same purpose.

To process the export and import directives, the compiler typically examines the import directives to obtain a list of potentially external references. These references are then examined by the compiler to determine their validity, to the extent that is possible at compile time. In Java, the `import` directives serve to initialize the symbol table so that references to abbreviated classes (`String` for `java.lang.String`) can be resolved.

8.4.5 Altered Search Rules

Pascal's `with` statement is a good example of a feature that alters the way in which symbols are found in the symbol table. If a Pascal program contains the phrase `with R do S`, then the statements in `S` first try to resolve references within the record `R`. If no valid reference is found in record `R`, then the symbol table is searched as usual. Inside the `with` block, fields that would otherwise be invisible are made visible. This feature prevents the programmer from frequently restating `R` inside `S`. Moreover, the compiler can usually generate faster code, since it is likely that record `R` is mentioned frequently in `S`.

Forward references also affect a symbol table's search rules. Consider a set of recursive data structures or methods. In the program, the set of definitions must be presented in some linear order. It is inevitable that a portion of the program will reference a definition that has not yet been processed. Forward references suspend the compiler's skepticism concerning undeclared symbols. A forward reference is essentially a promise that a complete definition will eventually be provided.

Some languages require that forward references be announced. In C, it is considered good style to *declare* an undefined function so that its types are known at forward references. In fact, some compilers require such declarations. On the other hand, a C structure may contain a field that is a pointer to itself. For example, each element in a linked list contains a pointer to another element. It is customary to process such forward references in two passes. The first pass makes note of types that should be checked in the second pass.

Summary

Although the interface for a symbol table is quite simple, the details underlying a symbol table's implementation play a significant role in the performance of the symbol table. Most modern programming languages are statically scoped. The symbol table organization presented in this chapter efficiently represents scope-declared symbols in a block-structured language. Each language places its own requirements on how symbols can be declared and used. Most languages include

rules for symbol promotion to a global scope. Issues such as inheritance, overloading, and aggregate data types should be considered when designing a symbol table.

Do not copy

Exercises

1. Consider a program in which the variable `x` is declared as a method's parameter *and* as one of the method's local variables. A programming language includes **parameter hiding** if the local variable's declaration can mask the parameter's declaration. Otherwise, the situation described in this exercise results in a multiply defined symbol. With regard to the symbol table interface presented in Section 8.1.2, explain the implicit scope-changing actions that must be taken if the language calls for
 - (a) parameter hiding
 - (b) no parameter hiding.
2. Code-generation sequences often require the materialization of temporary storage. For example, a complex arithmetic expression may require temporaries to hold intermediate values. Hopefully, machine registers can accommodate such short-lived values. However, compilers must sometimes plan to **spill** values out of registers into temporary storage.
 - (a) Where are symbol table actions needed to allocate storage for temporaries?
 - (b) How can implicit scoping serve to limit the effective lifetime of temporaries?
3. Consider the following C program.

```
int func() {
    int x, y;
    x = 10;
    {
        int x;
        x = 20;
        y = x;
    }
    return(x * y);
}
```

The identifier `x` is declared in the method's outer scope. Another declaration occurs within the nested scope that assigns `y`. For C, the nested declaration of `x` could be regarded as a declaration of some other name, provided that the inner scope's reference to `x` is appropriately renamed. Design an AST pass that

- Renames and moves nested variable declarations to the method's outermost scope

- Appropriately renames symbol references to preserve the meaning of the program
4. Recalling the discussion of Section 8.3.1, suppose that *all* active names are contained in a single, ordered list. An identifier would appear k times in the list if there are currently k active scopes that declare the identifier.
 - (a) How would you implement the methods defined in Section 8.1.2 so that that RETRIEVESYMBOL finds the appropriate declaration of a given identifier?
 - (b) Explain why the lookup time does or does not remain $O(\log(n))$ for a list of n entries.
 5. Design and implement a symbol table using the ordered list data structure suggested in Section 8.3.1. Discuss the time required to perform each of the methods defined in Section 8.1.2.
 6. Some languages contain names whose suffix is sufficient to locate the name. In Java, the classes in the package `java.lang` are available, so that the `java.lang.Integer` class can be referenced simply as `Integer`. This feature is also available for any explicitly imported classes. Similarly, Pascal's `with` statement allows field names to be abbreviated in the applicable blocks.

Design a name space that supports the efficient retrieval of such abbreviated names, under the following conditions. Be sure to document any changes you wish to make to the symbol table interface given in Section 8.1.2.
 7. Section 8.4.1 states that some languages allow a series of field references to be abbreviated, providing the abbreviation can uniquely locate the desired field. Certainly, the last field of such a reference must appear in the abbreviation. Moreover, the first field is necessary to distinguish the reference from other instances of the same type. We therefore assume that the first and last fields must appear in an abbreviation.

As an example, the reference `a.b.c.d` could be abbreviated `a.d` if records `a` and `b` do not contain their own `d` field.

Design an algorithm to allow such abbreviations, taking into consideration that the first and last fields of the reference cannot be omitted. Integrate your solution into RETRIEVESYMBOL in Section 8.3.3.
 8. Consider a language whose field references are the reverse of those found in C and Pascal. A C reference to `a.b.c.d` would be specified as `d.c.b.a`. Describe the modifications you would make to your symbol table to accommodate such languages.
 9. Program a symbol table manager using the interface described in Section 8.1.2 and the implementation described in Section 8.3.3.

10. Program a symbol table manager using the interface described in Section 8.1.2. Maintain the symbol table using *red-black trees* [?].
11. Program a symbol table manager using the interface described in Section 8.1.2. Maintain the symbol table using *splay trees* [?].
12. Describe how you would use the symbol table interface given in Section 8.1.2 to localize the declaration and effects of a loop iteration variable. As an example, consider the variable *i* in the Java-like statement

```
for (int i=1; i<10; ++i) {...}.
```

In this exercise, we seek the following.

- The declaration of *i* could not possibly conflict with any other declaration.
 - The effects on this *i* are confined to the body of the loop. That is, the scope of *i* includes the expressions of the `for` statement as well as its body, represented above as `{...}`. The value of the loop's iteration variable is undefined when the loop exits.
13. As mentioned in Section 8.4.2, C allows the same identifier to appear as a struct name, a label, and an ordinary variable name. Thus, the following is a valid C program:

```
main() {
    struct xxx {
        int a,b;
    } c;
    int xxx;

    xxx:
        c.a = 1;
}
```

In C, the structure name never appears without the terminal `struct` preceding it. The label can only be used as the target of a `goto` statement.

Explain how to use the symbol table interface given in Section 8.1.2 to allow all three varieties of `xxx` to coexist in the same scope.

14. Using the symbol table interface given in Section 8.1.2, describe how to implement records (`structs` in C) under each of the following assumptions.
 - (a) All records and fields are entered in a single symbol table.
 - (b) A record is represented by its own symbol table, whose contents are the record's subfields.

15. Describe how to implement typenames (`typedef` in C) in a standard symbol table. Consider the following program fragment:

```
typedef
    struct {
        int x,y;
    } *Pair;

Pair *(pairs[23]);
```

The `typedef` establishes `Pair` as a typename, defined as a *pointer* to a record of two integers. The declaration for `pairs` uses the typename, but adds one more level of indirection to an array of 23 `Pairs`. Your design for `typedef` must be accommodate further type construction using `typedefs`.

16. Each string in Figure 8.4 occupies its own space in the string buffer. Suppose the strings were added in the following order: `i`, `input`, and `putter`. If the strings could share common characters, then these three strings can be represented using only 8 character slots.
- Design a symbol table string-space manager that allows strings to overlap, retaining the representation of each string as an offset and length pair, as described in Section 8.3.2.
17. The two data structures most commonly used to implement symbol tables in production compilers are binary search and hash tables. What are the advantages and disadvantages of using these data structures for symbol tables?
18. Describe two alternative approaches to handling multiple scopes in a symbol table, and list the actions required to open and close a scope for each alternative. Trace the sequence of actions that would be performed for each alternative during compilation of the program in Figure 8.1.
19. Figure 8.6 provides code to create a single symbol table for all scopes. Recalling the discussion of Section 8.2.2, an alternative approach segregates symbols by scope, as shown in Figure 8.8. Modify the code of Figure 8.6 to manage a stack of symbol tables, one for each active scope. Retain the symbol table interface as defined in Section 8.1.2.

9

Semantic Analysis: Simple Declarations and Expressions

There are three major sections to this chapter. The first section presents some basic concepts related to processing declarations, including the ideas that a symbol table is used to represent information about names and that recursive traversal of an abstract syntax tree can be used to interpret declarations and use the information they provide to check types. The second section describes the tree-traversal “semantic” actions necessary to process a simple declarations of variables and types as found in commonly used programming languages. The third section includes similar semantic action descriptions for type checking abstract syntax trees corresponding to simple assignment statements. We continue to use semantic action descriptions in the next three chapters as a mechanism to discuss the compilation of more complex language features. While this presentation is based on the features of particular languages, the techniques introduced are general enough for use in compiling a wide variety of languages.

9.1 Declaration Processing Fundamentals

9.1.1 Attributes in the Symbol Table

In Chapter 8, symbol tables were presented as a means of associating names with some *attribute information*. We did not specify what kind of information was included in the attributes associated with a name or how it was represented. These topics are considered in this section.

The attributes of a name generally include anything the compiler knows about it. Because a compiler's main source of information about names is declarations, attributes can be thought of as internal representations of declarations. Compilers do generate some attribute information internally, typically from the context in which the declaration of a name appears or, when use is an implicit declaration, from a use of the name. Names are used in many different ways in a modern programming language, including as variables, constants, types, classes, and procedures. Every name, therefore, will not have the same set of attributes associated with it. Rather, it will have a set of attributes corresponding to its usage and thus to its declaration.

Attributes differing by the kind of name represented are easily implemented in either object-oriented or conventional programming languages. In a language like Java, we can declare an `Attributes` class to serve as the generalized type and then create subclasses of `Attributes` to represent the details of each specialized version needed. In Pascal and Ada, we can accomplish the same thing using variant records; in C, unions provide the necessary capability.

Figure 9.1 illustrates the style we will use to diagram data structures needed by a compiler to represent attributes and other associated information. In these diagrams, type names are shown in italics (e.g., *VariableAttributes*), while instance variable names and values are shown in the regular form of our program font (e.g., `VariableName`). This figure illustrates the information needed to represent the attributes of names used for variables and types. It includes two different subclasses of `Attributes` necessary for describing what the compiler knows about these two different kinds of names.

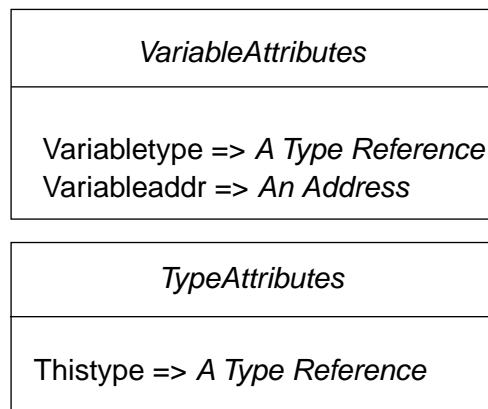


Figure 9.1 Attribute Descriptor Objects

Arbitrarily complex structures, determined by the complexity of the information to be stored, may be used for attributes. Even the simple examples used here include references to other structures that are used to represent type information.

As we outline the semantic actions for other language features, we will define a similar attribute structures for each.

9.1.2 Type Descriptor Structures

Among the attributes of almost any name is a type, which is represented by a *type reference*. A type reference is a reference to a `TypeDescriptor` object. Representing types presents a compiler writer with much the same problem as representing attributes: There are many different types whose descriptions require different information. As illustrated by the structures in Figure 9.2 by several subtypes of `TypeDescriptor`, we handle this requirement just as we did for the `Attributes` structures in the previous figure, with references to other type descriptor structures or additional information (such as the symbol table used to define the fields of a record), as necessary.

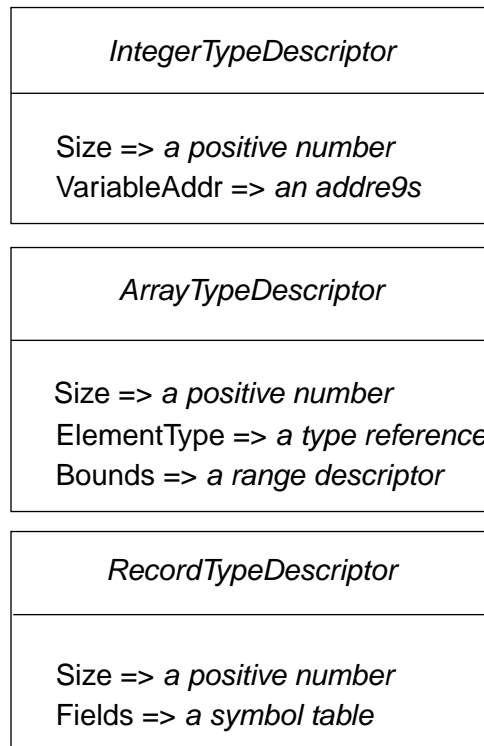


Figure 9.2 Type Descriptor Objects

The kinds of representation shown in this figure are crucial in handling the features of virtually all modern programming languages, which allow types to be constructed using powerful composition rules. Using this technique rather than some kind of fixed tabular representation also makes the compiler much more flexible in what it can allow a programmer to declare. For example, using this approach, there is no reason to have an upper bound on the number of dimensions allowed for an array or the number of fields allowed in a record or class. Such limitations in early languages like FORTRAN stem

purely from implementation considerations. We generally favor techniques that enable a compiler to avoid rejecting a legal program because of the size of the program or some part of it. Dynamic linked structures like the type descriptor structure are the basis of such techniques, as will be seen in this chapter and Chapter 14.

9.1.3 Type Checking Using an Abstract Syntax Tree

Given that we have a program represented by an abstract syntax tree, we can implement declaration processing and type checking (the semantic analysis task of our compiler) by a recursive walk through the tree. The attraction of this approach is that such a tree traversal can be specified quite easily through simple description of the actions to be performed when each kind of tree node is encountered. When each tree node is represented as an object, the necessary action can be expressed as a commonly-named method defined for all alternative subtypes of the node type. We will use the name `Semantics` for this method that performs the semantic analysis task.

If the implementation language does not support objects, this recursive traversal can be performed by a single routine called `Semantics`. Its body is most likely organized as a large case or switch statement with an alternative for each kind of node in an abstract syntax tree.

In the rest of this chapter and the three following chapters, we will describe the actions to be performed for each kind of tree node used to represent individual language features. Our actions can be interpreted as method bodies or switch/case alternatives, depending on the implementation most familiar to the reader.

Once execution of the tree traversal defined by `Semantics` is completed, the analysis phase of the compiler is complete. A second traversal can then be performed to synthesize intermediate code or target code. (A tree optimization pass could be done prior to code generation, particularly in the case where target code is to be generated.) We will define this second traversal similarly via methods called `CodeGen`. It has a structure much like that of the `Semantics` pass and we will similarly define it by defining the individual actions for the nodes corresponding to each language features.

Assume that a program is represented by an abstract syntax tree rooted in a `Program_Node` that includes as components references to two subtrees: one for a list of declarations and another for a list of statement. The common structure of the `Semantics` and `CodeGen` actions for this `Program_Node` are seen in Figure 9.3. Each simply invokes an equivalent operation on each of its subtrees.

```
PROGRAM_NODE.SEMANTICS( )
1.  Declarations.Semantics()
2.  Statements.Semantics()

PROGRAM_NODE.CODEGEN( )
1.  Declarations.CodeGen()
2.  Statements.CodeGen()
```

Figure 9.3 Traversing Program Trees

The `Semantics` and `CodeGen` actions for the lists of declarations and statements would likewise invoke the `Semantics` and `CodeGen` actions for each of the declarations and statements in the lists. Our primary focus in the rest of this chapter and the following chapters will be the actions needed to implement individual declarations and statements, as well as their components, particularly expressions.

9.2 Semantic Processing for Simple Declarations

We begin by examining the techniques necessary to handle declarations of variables and scalar types. Structured types will be considered in Chapter 13.

9.2.1 Simple Variable Declarations

Our study of declaration processing begins with simple variable declarations, those that include only variable names and types. (We will later consider the possibility of attaching various attributes to variables and providing an initial value.) Regardless of the exact syntactic form used in a particular language, the meaning of such declarations is quite straightforward: the identifiers given as variable names are to be entered into the symbol table for the current scope as variables. The type that is part of the declaration must be a component of the attributes information associated with each of the variables in the symbol table.

The abstract syntax tree built to represent a single variable declaration with the type defined by a type name is shown in Figure 9.4. (We will consider more general type specifications later.) Regardless of language syntax details, like whether the type comes before or after the variable name, this abstract syntax tree could be used to represent such declarations. If the declaration syntax includes a list of identifiers instead of only a single one, we can simply build a list of `VarDecl_Nodes`.

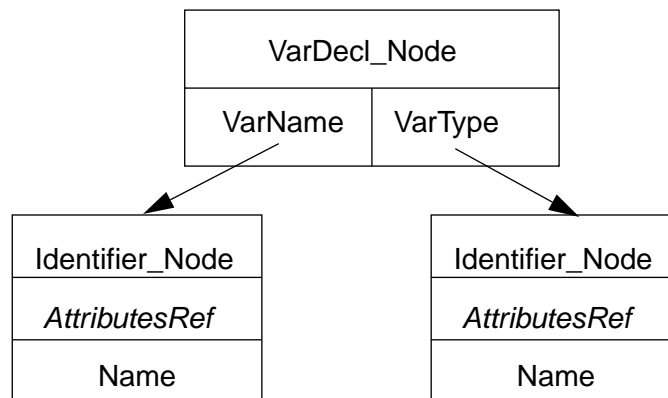


Figure 9.4 AST for a Variable Declaration

The **Semantics** action for variable declarations illustrates use of specialized semantic routines to process parts of the abstract syntax tree. As seen in this simple case, an **Identifier_Node** can be used in multiple contexts in a syntax tree. So, to define some context-specific processing to be done on it (rather than “default” actions), we specify invocation of a specialized actions like **TypeSemantics**. **TypeSemantics** attempts to interpret the **Identifier_Node** as describing a type. In fact, we will use **TypeSemantics** to access the type specified by any arbitrary subtree used where we expect a type specification.

The **Semantics** action on a **VarDecl_Node**, shown in Chapter 9.5, implements the meaning that we have already described for a variable declaration. Namely, it enters a into the current symbol table and associates an **Attributes** descriptor with it, indicating that the name is a variable and recording its type.

```

VARDECL_NODE.SEMANTICS( )
1.  Enter VarName.Name in the current symbol table
2.  if it is already there
3.      then Produce an error message indicating a duplicate declaration
4.      else Associate a VariableAttributes descriptor with the Id
              indicating:
              - its type that provided by invoking
                VarType.TypeSemantics ( )

```

Figure 9.5 VARDECL_NODE.SEMANTICS

The two routines in Chapter 9.6 process the type name. The first, **TypeSemantics**, is invoked in step 4 of **VarDecl_Node.Semantics** to interpret the type name provided as part of the variable declaration. The second is just the basic semantic action that accesses the attribute information associated with any name in the symbol table.

9.2.2 Type Definitions, Declarations, and References

In the previous section, we saw how type name references were processed, since they were used in the semantic processing for variable declarations. We now consider how type declarations are processed to create the structures accessed by type references. A type declaration in any programming language includes a name and a description of the corresponding type. The abstract syntax tree shown in Figure 9.4 can be used to represent such a declaration regardless of its source-level syntax. The **Semantics** action to declare a type (Figure 9.8) is similar for that used for declaring a variable: the type identifier must be entered into the current symbol table and an **Attributes** descriptor associated with it. In this case the **Attributes** descriptor must indicate that the identifier names a type and must contain a reference to a **TypeDescriptor** for the type it names.

IDENTIFIER_NODE.TYPE.SEMANTICS()

1. this_node.Semantics ()
2. **if** AttributesRef indicates that Name is a type name
3. **then** **return** the value of AttributesRef.ThisType()
4. **else** Produce an error message indicating that Name cannot be interpreted as a type
5. **return** ErrorType

IDENTIFIER_NODE.SEMANTICS()

1. Look Name up in the symbol table
2. **if** it is found there
3. **then** Set AttributesRef to the Attributes descriptor associated with Name
4. **else** Produce an error message indicating that Name has not been declared
5. Set AttributesRef to null

Figure 9.6 IDENTIFIER_NODE.TYPE.SEMANTICS

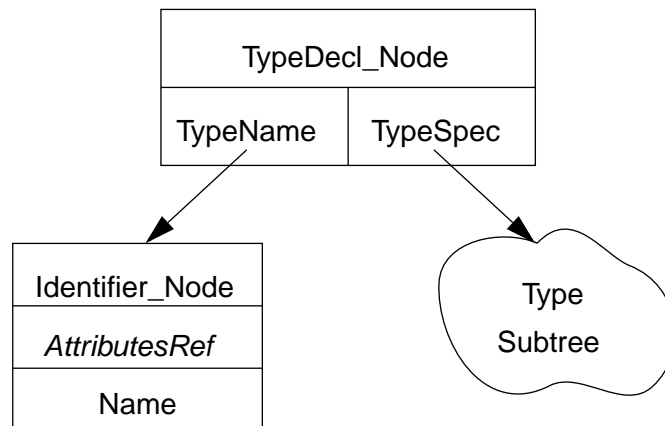


Figure 9.7 AST for a Type Declaration

Figure 9.7 and the pseudocode in Figure 9.8 leave unanswered the obvious question of what kind of subtree is pointed to by `TypeSpec`. This part of the abstract syntax tree defines the type represented by `TypeName`. Since type names are given to various types defined by a programmer, `TypeSpec` can point to a subtree that represents any form of type constructor allowed by the language being compiled. In Section 9.2.4, we will see how a simple type definition is processed. In Chapter 14, we will see how to compile to very commonly used type types: definitions of record and array types.

```

TYPEDECL_NODE.SEMANTICS( DeclType )
1.  Enter the TypeName.Name in the current symbol table
2.  if it is already there
3.      then Produce an error message indicating a duplicate dec-
         laration
4.      else Associate a TypeAttributes descriptor with the
         Name indicating:
         - the type referenced is that provided by invoking
           TypeSpec.TypeSemantics ()

```

Figure 9.8 TYPEDECL_NODE.SEMANTICS

By using `TypeSemantics` (rather than `Semantics`) to process the subtree referenced by `TypeSpec`, we require that the semantic processing for a type definition result in the construction of a `TypeDescriptor`. As we have just seen, the `Semantics` action for `TypeDecl_Node` then associates a reference to that `TypeDescriptor` with a type name. Notice, however, that things work just as well if a type definition rather than a type name is used in a variable declaration. In the previous section, we assumed that the type for a variable declaration was given by a type name. However, since we processed that name by calling `TypeSemantics`, it is actually irrelevant to the variable declaration pseudocode whether the type is described by a type name or by a type definition. In either case, processing the type subtree with `TypeSemantics` will result in a reference to a `TypeDescriptor` being returned. The only difference is that in one case (a name), the reference is obtained from the symbol table, while in the other (a type definition), the reference is to a descriptor created by the call to `TypeSemantics`.

Handling Type Declaration and Type Reference Errors. In the pseudocode for the `TypeSemantics` action for an identifier, we introduce the idea of a *static semantic check*. That is, we define a situation in which an error can be recognized in a program that is syntactically correct. The error in this case is a simple one: a name used as a type name does not actually name a type. In such a situation, we want our compiler to generate an error message. Ideally, we want to generate only one error message per error, even if the illegal type name is used many times. (Many compilers fall far short of this ideal!). The simplest way to accomplish this goal is to immediately stop the compilation, though it is also at least approachable for a compiler that tries to detect as many errors as possible in a source program.

Whenever `TypeSemantics` finds that a type error, it returns a reference to a special `TypeDescriptor` that we refer to as `ErrorType`. That particular value is a signal to the code that called `TypeSemantics` that an error has been detected and that an error message has already been generated. In this calling context, an explicit check may be made for `ErrorType` or it may be treated like any other `TypeDescriptor`. In the specific instance of the variable declaration pseudocode, the possibility of an `ErrorType` return can be ignored. It works perfectly well to declare variables with `ErrorType` as their type. In fact, doing so prevents later, unnecessary error messages. That is, if a variable is left

undeclared because of a type error in its declaration, each time it is used, the compiler will generate an undeclared variable error message. It is clearly better to declare the variable with `ErrorType` as its type in order to avoid these perhaps confusing messages. In later sections, we will see other uses of `ErrorType` to avoid the generation of extraneous error messages.

Type Compatibility. One question remains to be answered: Just what does it mean for types to be the same or for a constraint (as used in Ada) to be compatible with a type? Ada, Pascal, and Modula-2 have a strict definition of type equivalence that says that every type definition defines a new, distinct type that is incompatible with all other types. This definition means that the declarations

```
A, B : ARRAY (1..10) OF Integer;
C, D : ARRAY (1..10) OF Integer;
```

are equivalent to

```
type Type1 is ARRAY (1..10) OF Integer;
A, B : Type1;
type Type2 is ARRAY (1..10) OF Integer;
C, D : Type2;
```

A and B are of the same type and C and D are of the same type, but the two types are defined by distinct type definitions and thus are incompatible. As a result, assignment of the value of C to A would be illegal. This rule is easily enforced by a compiler. Since every type definition generates a distinct type descriptor, the test for type equivalence requires only a comparison of pointers.

Other languages, most notably C, C++ and Java, use different rules to define type equivalence, though Java does use name equivalence for classes. The most common alternative is to use *structural* type equivalence. As the name suggests, two types are equivalent under this rule if they have the same definitional structure. Thus `Type1` and `Type2` from the previous example would be considered equivalent. At first glance, this rule seems like a much more useful choice because it is apparently more convenient for programmers using the language. However, counterbalancing this convenience is the fact the structural type equivalence rule makes it impossible for a programmer to get full benefit from the concept of type checking. That is, even if a programmer wants the compiler to distinguish between `Type1` and `Type2` because they represent different concepts in the program despite their identical implementations, the compiler is unable to do so.

Structural equivalence is much harder to implement, too. Rather than being determined by a single pointer comparison, a parallel traversal of two type descriptor structures is required. The code to do such a traversal requires special cases for each of the type descriptor alternatives. Alternatively, as a type definition is processed by the semantic checking pass through the tree, the type being defined can be compared against previously defined types so that equivalent types are represented by the same data structure, even though they are defined separately. This technique allows the type equivalence test to be implemented by a pointer comparison, but it requires an indexing mechanism that makes it possible to tell during declaration processing whether each newly defined type is equivalent to *any* previously defined type.

Further, the recursion possible with pointer types poses subtle difficulties to the implementation of a structural type equivalence test. Consider the problem of writing a routine that can determine whether the following two Ada types are structurally equivalent, given the following example. (Access types in Ada roughly correspond to pointers in other languages.)

```
type A is access B;
type B is access A;
```

Even though such a definition is meaningless semantically, it is syntactically legal. Thus a compiler for a language with structural type equivalence rules must be able to make the appropriate determination — that A and B are equivalent. If parallel traversals are used to implement the equivalence test, the traversal routines must “remember” which type descriptors they have visited during the comparison process in order to avoid an infinite loop. Suffice it to say that comparing pointers to type descriptors is much simpler!

9.2.3 Variable Declarations Revisited

The variable declaration can be more complex than the simple form shown in Section 9.2.1. Many languages allow various modifiers, such as `constant` or `static`, to be part of a variable declaration. Visibility modes like `public`, `private` and `protected` may also be part of a declaration, particularly in object-oriented languages. Finally, an initial value might be specified. The abstract syntax tree node for a variable declaration obviously must be more complex than the one in Section 9.4 to encompass all of these possibilities. Thus we need a structure like the one we see in Figure 9.9.

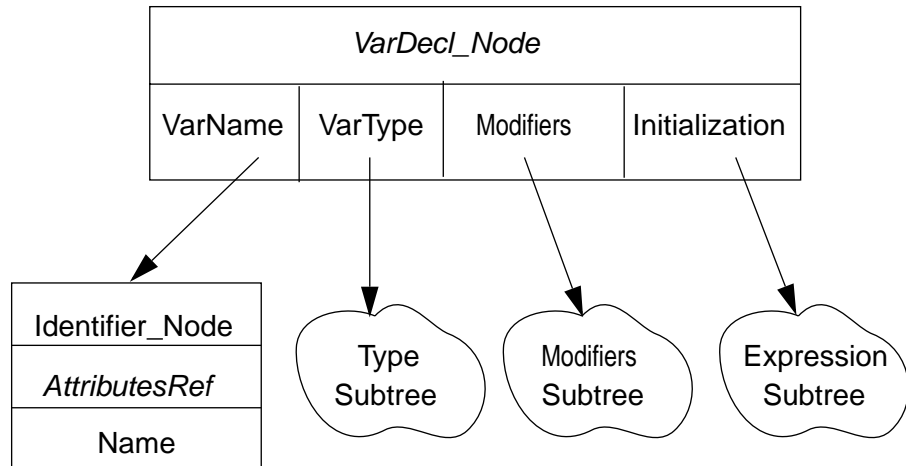


Figure 9.9 AST for a Complete Variable Declaration

Both the modifiers and initialization parts are optional; in the case that neither are present, we are left with the same information as in the simple version of this node seen

in Section 9.2.1. When present, the expression subtree referenced by `Initialization` may be a literal constant whose value can be computed at compile time or its value may be determined by an expression that cannot be evaluated until run-time. If it has a compile-time value and `constant` is one of the `Modifiers`, the “variable” being declared is actually a constant that may not even require run-time space allocation. Those that do require space at run-time act much like variables, though the `Attributes` descriptor of such a constant must include an indication that it may not be modified. We will leave the structure of the `Modifiers` subtree unspecified, vary according to the specification of the particular language being compiled. For a language that allows only a single modifier (`constant`, perhaps), there need be no subtree since a single boolean component in the `VarDecl_Node` will suffice. At the other extreme, Java allows several modifiers to be present along with a name being declared, necessitating use of a list or some other structure with sufficient flexibility to handle all of the possible combinations of modifiers.

The `Semantics` action for this extended `VarDecl_Node` (in Figure 9.10) handles the specified type and the list of variable or constant names much as they were handled for the simpler `VarDecl_Node` in Section 9.2.1.

9.2.4 Enumeration Types

An enumeration type is defined by a list of distinct identifiers denoting its values; each identifier is a constant of the enumeration type. Enumeration constants are ordered by their position in the type definition and are represented internally by integer values. Typically, the value used to represent the first identifier is zero, and the value for each subsequent identifier is one more than that for its predecessor in the list (though some languages do allow a programmer to specify the values used to represent enumeration literals).

The abstract syntax tree representation for an enumeration type definition is shown in Figure 9.11. The `TypeSemantics` action for processing an `EnumType_Node`, like those for processing records and arrays, will build a `TypeDescriptor` that describes the enumeration type. In addition, each of the identifiers used to define the enumeration type is entered into the current symbol table. Its attributes will include the fact that it is an enumeration constant, the value used to represent it, and a reference to the `TypeDescriptor` for the enumeration type itself. The type will be represented by a list of the `Attributes` records for its constants. The required specializations of `Attributes` and `TypeDescriptor` are illustrated in Figure 9.12 and the pseudocode for the enumeration type semantics action is in Figure 9.13.

Figure 9.14 shows the `TypeDescriptor` that would be constructed to represent the enumeration type defined by

```
(red, yellow, blue, green)
```

The `Size` of the enumeration type is set during the `CodeGen` pass through the tree to the size of an integer (or perhaps just a byte), which is all that is needed to store an enumeration value.

```

VARDECL_NODE.SEMANTICS( )
1.  Local Variables: DeclType, InitType
2.  Set DeclType To VarType.TypeSemantics()
3.  if Initialization is not a null pointer
4.      then Set Inittype to Initialization.ExprSemantics ( )
5.          if not Assignable (InitType, DeclType)
6.              then Generate an appropriate error message
7.                  Set DeclType to ErrorType
8.          else • Initialization Is Null
9.              Check that Constant is not among the modifiers
10.         Enter VarName.Name in the current symbol table
11.         if it is already there
12.             then Produce an error message indicating a duplicate declaration
13.         elseif Constant is among the modifiers
14.             then Associate a ConstAttributes descriptor with it indicating:
15.                 - Its type is DeclType
16.                 - Its value is defined by Initialization (a pointer to an expression tree)
17.         else • It is a variable
18.             Associate a VariableAttributes descriptor with it indicating:
19.                 - Its type is DeclType
20.                 - it modifiers are those specified by Modifiers
21.             • Any initialization will be handled by the code generation pass

```

Figure 9.10 VARDECL_NODE.SEMANTICS (VERSION 2)

9.3 Semantic Processing for Simple Names and Expressions: An Introduction to Type Checking

The semantic processing requirements for names and expressions can be easily understood in the context of an assignment statement. As illustrated in the AST representations shown in Figure 9.15, an assignment is defined by a name that specifies the target of the assignment and an expression that must be evaluated in order to provide the value to be assigned to the target. The name may be either a simple identifier or a qualified identifier, the latter in the case of a reference to a component of a data structure or object.

In this chapter, we will only consider the case where the target is a simple identifier, which can be represented by the same `Identifier_Node` that has been used in several other contexts in this chapter. The simplest form of an expression is also just a name (as seen in the assignment statement `A = B;`) or a literal constant (as in `A = 5;`). The section that follows will present the `Semantics` actions necessary to handle these simple

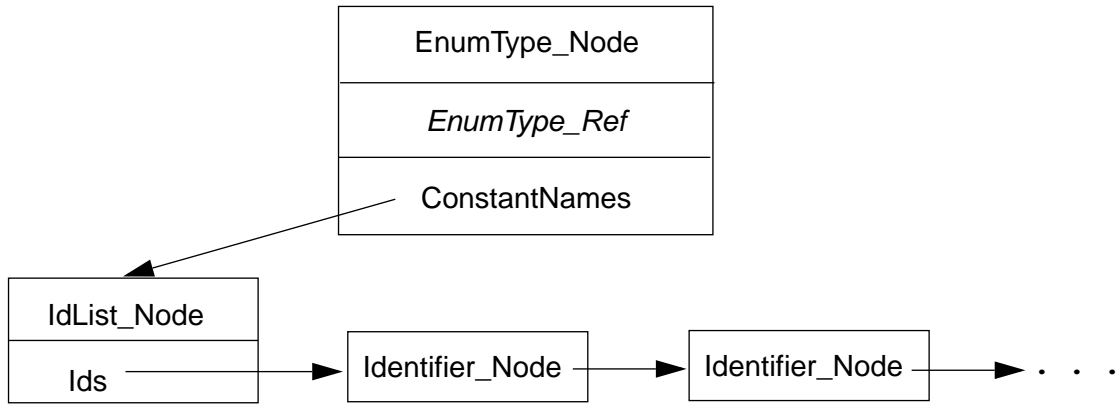


Figure 9.11 AST for an Enumeration Type Definition

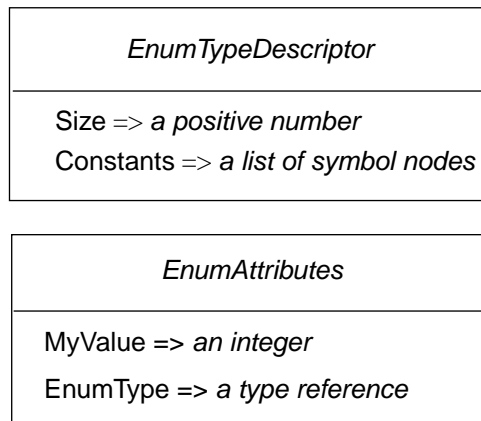


Figure 9.12 Type and Attribute Descriptor Objects for Enumerations

kinds of expressions and assignments. Section 9.3.2 will deal with expressions involving unary and binary operators and in Chapter 14 the techniques needed to compile simple record and array references will be presented.

9.3.1 Handling Simple Identifiers and Literal Constants

The syntax trees for these assignments are the simple ones seen in Figure 9.9. The **Semantics** actions needed to handle these assignment statements are equally simple. However, thoughtful examination of the tree for $A = B$ leads to the observation that the two identifiers are being used in rather different ways. The A stands for the address of A , the location that receives the assigned value. The B stands for the value at the address associated with B . We say that the A is providing an L-value (an address) because this the

```

ENUMTYPE_NODE.TYPESEMANTICS( )
1. [Local Variable: Nextvalue, Enumtype_ref]
2. Create An Enumtypedescriptor For The New Enumeration
   Type With:
3.     Constants Pointing To An Empty List Of Symbols
4. Set Enumtype_ref To Point To This Enumtypedescriptor
5. Set Nextvalue To 0
6. for Each Of The Identifiers In The Constantnames List
7.     do Enter The Identifier In The Current Symbol Table
8.     Associate An Enumattributes Descriptor With It
   Indicating:
9.     Its Type Is The Typedescriptor Pointed To
   By Enumtype_ref
10.    Its Value Is Nextvalue
11.    Increment Nextvalue
12. return Enumtype_ref

```

Figure 9.13 ENUMTYPE_NODE.TYPESEMANTICS

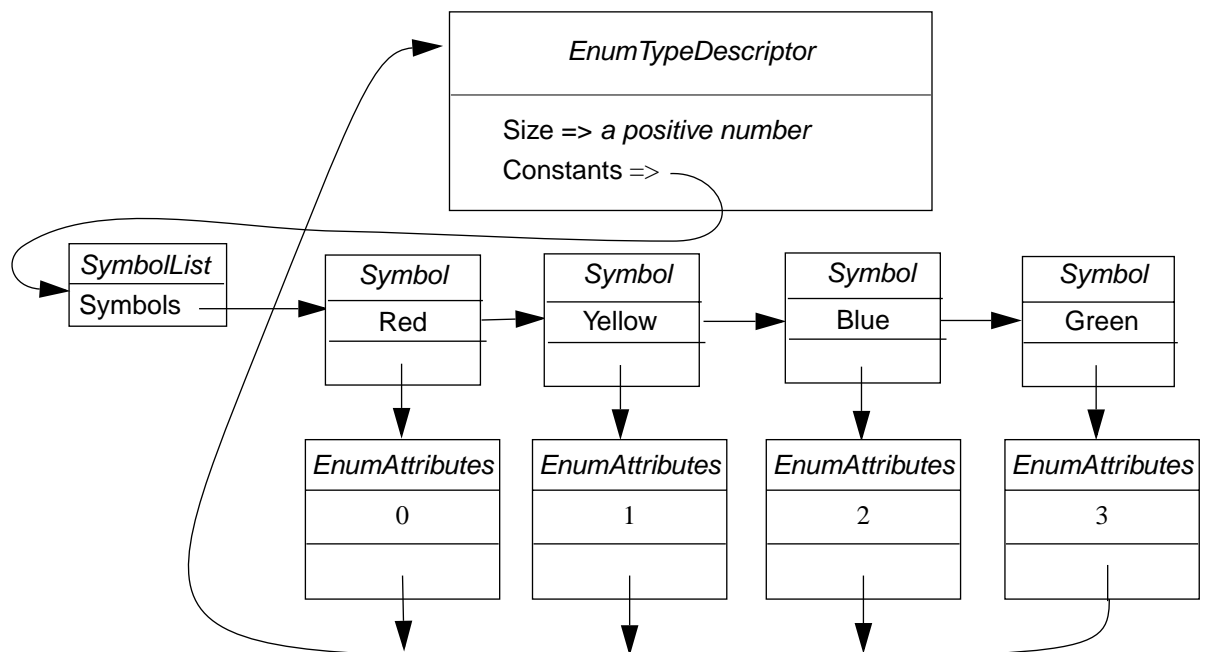


Figure 9.14 Representation of an Enumeration Type

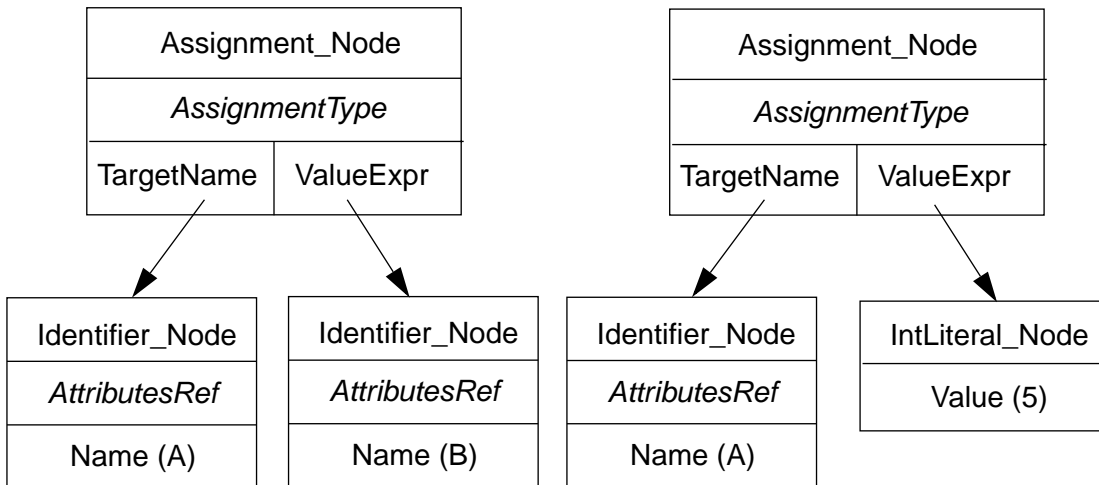


Figure 9.15 AST Representation of an Assignment Statement

meaning of an identifier on the left-hand-side of an assignment statement. Similarly, we say that B denotes an R-value (a value or the contents of an address) because that is how we interpret an identifier on the right-hand-side of an assignment. From these concepts, we can infer that when we consider more general assignment statements, the `TargetName` subtree must supply an L-value, while the `ValueExpr` subtree must supply an R-value.

Since we would like to be able to treat all occurrences of `Identifier_Node` in an expression tree consistently, we will say that they always provide an L-value. When an identifier is used in a context where an R-value is required, the necessary conversion must be implicitly performed. This requirement actually has no implication on type checking in most languages, but will require that appropriate code be generated based on the context in which a name is used.

The major task of the `Assignment_Node Semantics` action, seen in Figure 9.20, involves obtaining the types of the components of the assignment statement and checking whether they are compatible according to the rules of the language. First, the target name is interpreted using a specialized version of `Semantics` called `LvalueSemantics`. It insures that the target designates an assignable entity, that is, one that produces an L-value. Named constants and constant parameters look just like variables and can be used interchangeably when an R-value is required, but they are not acceptable target names. A semantic attribute `AssignmentType` is used to record the type to be assigned after this determination has been made so that the type is easily available to the `CodeGen` actions that will be invoked during the next pass.

A second specialized version of `Semantics`, called `ExprSemantics`, is used to evaluate the type of an expression AST subtree. Figure 9.17 includes two versions of `ExprSemantics` (and more will be seen shortly, for processing binary and unary expressions). The one for `Identifier_Node` makes use of the general `Identifier_Node.Semantics`

```

ASSIGNMENT_NODE.SEMANTICS ( )
1. [Local variables: LeftType, RightType]
2. Set LeftType to TargetName.LvalueSemantics ( )
3. Set RightType to ValueExpr.ExprSemantics ( )
4. if RightType is assignable to LeftType
5.     then Set AssignmentType to LeftType
6.     else Generate an appropriate error message
7.         Set AssignmentType to ErrorType

IDENTIFIER_NODE.LVALUESEMANTICS ( )
1. Call this_node.Semantics ( )
2. if AttributesRef indicates that Name denotes a L-value
3.     then return the type associated with Name
4.     else Produce an error message indicating that Name cannot
           be interpreted as a variable
5.     return ErrorType

```

Figure 9.16

tics action defined earlier in this chapter (in Section 9.2). It simply looks up *Name* in the symbol table and sets the semantic attribute *AttributesRef* to references *Name*'s attributes. Note that both *LvalueSemantics* and *ExprSemantics* are much like *TypeSemantics*, which was defined along with *Identifier_Node.Semantics* in Section 9.2.2. They differ only in the properties they require of the name they are examining. The *ExprSemantics* action for an integer literal node is quite simple. No checking is necessary, since a literal is known to be an R-value and its type is immediately available from the literal node itself.

```

IDENTIFIER_NODE.EXPRSEMANTICS ( )
1. Call this_node.Semantics ( )
2. if AttributesRef indicates that Name denotes an R-value (or an
   L-value which can be used to provide one)
3.     then return the type associated with Name
4.     else Produce an error message indicating that Name cannot
           be interpreted as a variable
5.     return ErrorType

INTLITERAL_NODE.EXPRSEMANTICS ( )
1. return IntegerType

```

Figure 9.17 EXPRSEMANTICS for Identifiers and Literals

We will see the type checking alternatives for more complex names, such as record field and array element references, in a later chapter. They will be invoked by the same calls from the `Semantics` actions for an `Assignment_Node` and they will return types just as in the simple cases we have just seen. Thus the processing done for an assignment can completely ignore the complexity of the subtree that specifies the target of the assignment, just as it need not consider the complexity of the computation that specifies the value to be assigned.

9.3.2 Processing Expressions

In the previous section, we saw two examples of the general concept of an expression, namely, a simple identifier and a literal constant. More complex expressions in any programming language are constructed using unary and binary operators. The abstract syntax tree representations for expressions follow naturally from their common syntactic structure, as illustrated in Figure 9.18. The nodes referenced by `LeftExpr`, `RightExpr` and `SubExpr` in the figure can be additional expression nodes or they can be from among the group that specifies operands (that is, literals, identifiers, etc.). As discussed in the previous section, the semantic processing and code generation actions for all of these nodes will behave in a consistent fashion, which will enable the actions for expression processing to be independent of concerns about the details of the expression's operands.

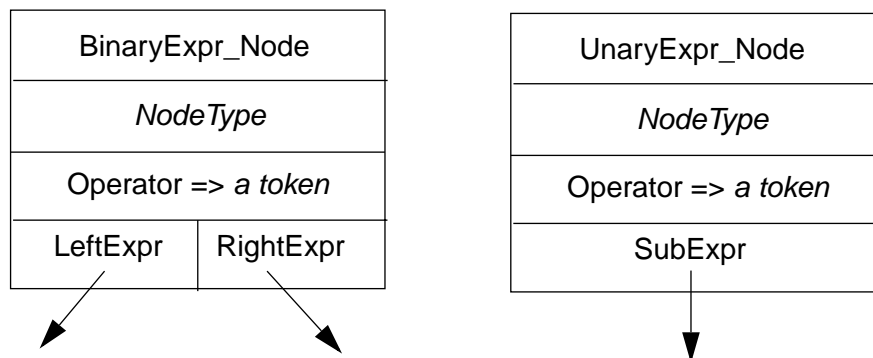


Figure 9.18 AST Representations for Expressions

The type checking aspects of handling expression nodes depend primarily on the types of the operands. Thus, both of the `ExprSemantics` actions below first invoke `ExprSemantics` on the operands of the expression. The next step is to find whether the operator of the expression is meaningful given the types of the operands. We illustrate this test as being done by the routines `BinaryResultType` and `UnaryResultType` in the pseudocode in Figure 9.19. These routines must answer this question based on the definition of the language being compiled. If the specified operation is meaningful, the type of the result of the operation is returned; otherwise, the result of the call must be `ErrorType`. To consider a few examples: adding integers is defined in just about all programming languages, with the result being an integer. Addition of an integer and a real will

produce a real in most languages, but in a language like Ada that does not allow implicit type conversions, such an expression is erroneous. Finally, comparison of two arithmetic values typically yields a boolean result.

```

BINARYEXPR_NODE.EXPRSEMANTICS ( )
1.  [Local variables: LeftType, RightType]
2.  Set LeftType to LeftExpr.ExprSemantics ( )
3.  Set RightType to RightExpr.ExprSemantics ( )
4.  Set NodeType to BinaryResultType (Operator, LeftType,
    RightType)
5.  return NodeType

UNARYEXPR_NODE.EXPRSEMANTICS ( )
1.  [Local variable: SubType]
2.  Set SubType to SubExpr.ExprSemantics ( )
3.  Set NodeType to UnaryResultType (Operator, SubType)
4.  return NodeType

```

Figure 9.19 EXPRSEMANTICS for Binary and Unary Expressions

9.4 Key Idea Summary

Section 9.1 presented three key ideas: a symbol table is used to associate names with a variety of information, generally referred to as attributes; types are a common, distinguished kind of attribute; and types and other attributes are represented by records (structures) with variants (unions) designed as needed to store the appropriate descriptive information. Section 9.1 also introduced the mechanism of recursive traversal by which **Semantics** and **CodeGen** passes over an abstract syntax tree can be used to type check and generate intermediate or target code for a program.

Section 9.2 described the **Semantics** actions for handling the abstract syntax tree nodes found in simple declarations during the declaration processing and type checking pass done by the **Semantics** traversal. The concept of using specialized semantics actions, such as **TypeSemantics** and **ExprSemantics** was introduced in this section as was the notion of using an **ErrorType** to deal with type errors in a way so as to minimize the number of resulting error messages.

Finally, Section 9.3 introduced type checking using abstract syntax trees for simple assignments as an example framework. The concept of interpreting names as L-values or R-values was described, along with its implications for AST processing.

10

Semantic Analysis: Control Structures and Subroutines

10.1 Semantic Analysis for Control Structures

Control structures are an essential component of all programming languages. With control structures a programmer can combine individual statements and expressions to form an unlimited number of specialized program constructs.

Some languages constructs, like the if, switch and case statement, provide for conditional execution of selected statements. Others, like while, do and for loops, provide for iteration (or repetition) of the body of a looping construct. Still other statements, like the break, continue, throw, return and goto statements force program execution to depart from normal sequential execution of statements.

The exact form of control structures differs in various programming languages. For example, in C, C++ and Java, if statements don't use a then keyword; in Pascal, ML Ada, and many other languages, then is required.

Minor syntactic differences are unimportant when semantic analysis is performed by a compiler. We can ignore syntactic differences by analyzing an **abstract syntax tree**. Recall that an abstract syntax tree (AST) represents the essential structure of a construct while hiding minor variations in source level representation. Using an AST, we can discuss the semantic analysis of fundamental constructs like conditional and looping statements without any concern about exactly how they are represented at the source level.

An important issue in analyzing Java control structures is **reachability**. Java requires that unreachable statements be detected during semantic analysis, with suitable error messages generated. For example, in the statement sequence

```
...; return; a=a+1; ...
```

the assignment statement must be marked as **unreachable** during semantic analysis.

Reachability analysis is **conservative**. In general, determining whether a given statement can be reached is very difficult. In fact it is impossible! Theoretical computer scientists have proven that it is **undecidable** whether a given statement is ever executed, even when we know in advance all the data a program will access (reachability is a variant of the famous halting problem first discussed in [Turing 1936]).

Because our analyses will be conservative, we will not detect all occurrences of unreachable statements. However, the statements we recognize as unreachable will definitely be erroneous, so our analysis will certainly be useful. In fact, even in languages like C and C++, which don't require reachability analysis, we can still produce useful warnings about unreachable statements that may well be erroneous.

To detect unreachable statements during semantic analysis, we'll add two boolean-valued fields to the ASTs that represent statements and statement lists. The first, `isReachable`, marks whether a statement or statement list is considered reachable. We'll issue an error message for any non-null statement or statement list for which `isReachable` is false.

The second field, `terminatesNormally`, marks whether a given statement or statement list is expected to terminate normally. A statement that terminates normally will continue execution "normally" with the next statement that follows. Some statements (like a break, continue or return) may force execution to proceed to a statement other than the normal successor statement. These statements are marked with `terminatesNormally = false`. Similarly, a loop may never terminate its iteration (e.g., `for (; ;) { a=a+1; }`). Loops that don't terminate (using a conservative analysis) also have their `terminatesNormally` flag set to false.

The `isReachable` and `terminatesNormally` fields are set according to the following rules:

- If `isReachable` is true for a statement list, then it is also true for the first statement in the list.
- If `terminatesNormally` is false for the last statement in a statement list, then it is also false for the whole statement list.
- The statement list that comprises the body of a method, constructor, or static initializer is always considered reachable (its `isReachable` value is true).
- A local variable declaration or an expression statement (assignment, method call, heap allocation, variable increment or decrement) always has `terminatesNormally` set to true (even if the statement has `isReachable` set to false). (This is done so that all the statements following an unreachable statement don't generate error messages).

- A null statement or statement list never generates an error message if its `isReachable` field is false. Rather, the `isReachable` value is propagated to the statement's (or statement list's) successor.
- If a statement has a predecessor (it is not the first statement in a list), then its `isReachable` value is equal to its predecessor's `terminatesNormally` value. That is, a statement is reachable if and only if its predecessor terminates normally.

As an example, consider the following method body

```
void example() {
    int a; a++; return; ; a=10; a=20; }
```

The method body is considered reachable, and thus so is the declaration of `a`. This declaration and the increment of `a` complete normally, but the `return` does not (see Section 10.1.4). The null statement following the `return` is unreachable, and propagates this fact to the assignment of 10, which receives an error message. This assignment terminates normally, so its successor is considered reachable.

In the succeeding sections we will study the semantic analysis of the control structures of `Java`. Included in this analysis will be whether the statement in question terminates normally. Thus we will set the `terminatesNormally` value for each kind of control statement, and from this successor statements will set their `isReachable` field.

Interestingly, although we expect most statements and statement lists to terminate normally, in functions (methods that return a non-void value) we **require** the method body to terminate abnormally. Because a function must return a value, it cannot return by “falling through” to the end of the function. It must execute a return of some value or throw an exception. These both terminate abnormally, so the method body must also terminate abnormally. After the body of a function is analyzed, the `terminatesNormally` value of the statement list comprising the body is checked; if it is not false, an error message (“Function body must exist with a return or throw statement”) is generated.

In analyzing expressions and statements, we must be aware of the fact that the constructs may throw an exception rather than terminate normally. `Java` requires that all checked exceptions be accounted for. That is, if a checked exception is thrown (see Section 10.1.6), then it **must** either be caught in a catch block or listed in a method's throw list.

To enforce this rule, we'll accumulate the checked exceptions that can be generated by a given construct. Each AST node that contains an expression or statement will have a `throwsSet` field. This field will reference a linked list of `throwItem` nodes. Each `throwItem` has fields `type` (the exception type that is thrown), and `next`, a link to the next `throwItem`. This list, maintained as a set (so that duplicate exceptions are removed), will be propagated as ASTs are analyzed. It will be used when catch blocks and methods and constructors are analyzed.

10.1.1 If Statements

The AST corresponding to an if statement is shown in Figure 10.1. An `IfNode` has three subtrees, corresponding to the condition controlling the if, the then statements and the else statements. The semantic rules for an if statement are simple—the condition must be a valid boolean-valued expression and the then and else statements must be semantically valid. An if statement terminates normally if either its then part or its else part terminates normally. Since null statements terminate trivially, an if-then statement (with a null else part) always terminates normally.

The `Semantics` method for an if is shown in Figure 10.2.

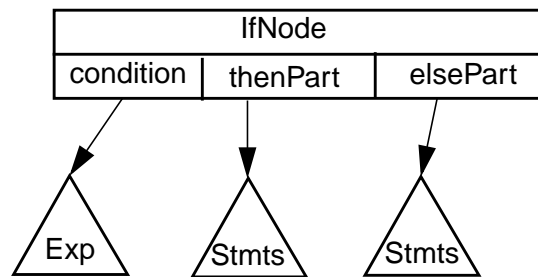


Figure 10.1 Abstract Syntax Tree for an if statement

Since `condition` must be an expression, we first call `ExprSemantics` to determine the expression’s type. We expect a boolean type. It may happen that the condition itself contains a semantic error (e.g., an undeclared identifier or an invalid expression). In this case, `ExprSemantics` returns the special type `errorType`, which indicates that further analysis of condition is unnecessary (since we’ve already marked the condition as erroneous). Any type other than `boolean` or `errorType` causes an “Illegal conditional expression” error message to be produced.

Besides the `condition` subtree, the `thenPart` and `elsePart` ASTs must be checked. This is done in lines 4 to 8 of the algorithm. The if statement is marked as completing normally if either the then or else part completes normally. The set of exceptions that an if statement can throw is the set union of throws potentially generated in the condition expression and the then and else statements. Recall that if an if statement has no else part, the `elsePart` AST is a `NullNode`, which is trivially correct.

As an example, consider the following statement

```
if (b) a=1; else a=2;
```

We first check the condition expression, `b`, which must produce a boolean value. Then the then and else parts are checked; these must be valid statements. Since assignment statements always complete normally, so does the if statement.

The modularity of our AST formulation is apparent here. The checks applied to control expression `b` are the same used for *all* expressions. Similarly, the checks applied to the then and else statements are those applied to *all* statements. Note

```

IfNode.Semantics( )
1.  ConditionType ← condition.ExprSemantics()
2.  if ConditionType ≠ boolean and ConditionType ≠ errorType
3.      then GenerateErrorMessage("Illegal conditional expression")
4.  thenPart.isReachable ← elsePart.isReachable ← true
5.  thenPart.Semantics()
6.  elsePart.Semantics()
7.  terminatesNormally ←
      thenPart.terminatesNormally or elsePart.terminatesNormally
8.  throwsSet ← union(condition.throwsSet, thenPart.throwsSet, elsePart.throwsSet)

```

Figure 10.2 Semantic Checks for an If Statement

too that nested if statements cause no difficulties—the same checks are applied each time an `IfNode` is encountered.

10.1.2 While, Do and Repeat Loops

The AST corresponding to a while statement is shown in Figure 10.3. A `whileNode` has two subtrees, corresponding to the condition controlling the loop and the loop body. The semantic rules for a while statement are simple. The condition must be a valid boolean-valued expression and the loop body must be semantically valid.

Reachability analysis must consider the special case that the control expression is a constant. If the control expression is false, then the statement list comprising the loop body is marked as unreachable. If the control expression is true, the while loop is marked as abnormally terminating (because of an infinite loop). It may be that the loop body contains a reachable break statement. If this is the case, semantic processing of the break will reset the loop's `terminatesNormally` field to true. If the control expression is non-constant, the loop is marked as terminating normally. The exceptions potentially thrown are those generated by the loop control condition and the loop body.

The method `EvaluateConstExpr` traverses an expression AST that is semantically valid to determine whether it represents a constant-valued expression. If the AST is a constant expression, `EvaluateConstExpr` returns its value; otherwise it returns null. The `Semantics` method for a while loop is shown in Figure 10.4.

As an example, consider

```

while (i >= 0) {
    a[i--] = 0; }

```

The control expression, `i >= 0`, is first checked to see if it is a valid boolean-valued expression. Since this expression is non-constant, the loop body is assumed reachable and the loop is marked as terminating normally. Then the loop body is checked for possible semantic errors.

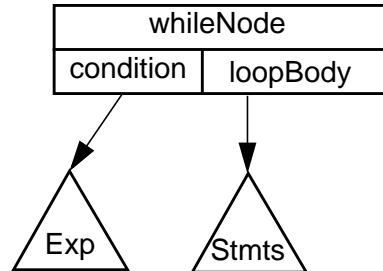


Figure 10.3 Abstract Syntax Tree for a While Statement

WhileNode.Semantics ()

1. ConditionType \leftarrow condition.ExprSemantics()
2. terminatesNormally \leftarrow true
3. loopBody.isReachable \leftarrow true
4. if ConditionType = boolean
5. then ConditionValue \leftarrow condition.EvaluateConstExpr()
6. if ConditionValue = true
7. then terminatesNormally \leftarrow false
8. elsif ConditionValue = false
9. then loopBody.isReachable \leftarrow false
10. elsif ConditionType \neq errorType
11. then GenerateErrorMessage("Illegal conditional expression")
12. loopBody.Semantics()
13. throwsSet \leftarrow union(condition.throwsSet, loopBody.throwsSet)

Figure 10.4 Semantic Checks a for While Statement

Do While and Repeat Loops. Java, C and C++ contain a variant of the while loop—the do while loop. A do while loop is just a while loop that evaluates and tests its termination condition *after* executing the loop body rather than before. The semantic rules for a do while are almost identical to those of a while. Since the loop body always executes at least once, the special case of a false control expression can be ignored. For non-constant loop control expressions, a do while loop terminates normally if the loop body does. Assuming the same AST structure as a while loop, the semantic processing appropriate for a do while loop is shown in Figure 10.5.

A number of languages, including Pascal and Modula 3, contain a repeat until loop. This is essentially a do while loop except for the fact that the loop is terminated when the control condition becomes false rather than true. The semantic rules of a repeat until loop are almost identical to those of the do while loop. The only change is that the special case of a non-terminating loop occurs when the control expression is false rather than true.


```

DoWhileNode.Semantics ( )
1.  loopBody.isReachable ← true
2.  loopBody.Semantics()
3.  ConditionType ← condition.ExprSemantics()
4.  if ConditionType = boolean
5.      then ConditionValue ← condition.EvaluateConstExpr()
6.          if ConditionValue = true
7.              then terminatesNormally ← false
8.              else terminatesNormally ← loopBody.terminatesNormally
9.  elsif ConditionType ≠ errorType
10.     then GenerateErrorMessage("Illegal conditional expression")
11.     throwsSet ← union(condition.throwsSet, loopBody.throwsSet)

```

Figure 10.5 Semantic Checks a for Do While Statement

10.1.3 For Loops

For loops are normally used to step an index variable through a range of values. However, for loops in **C**, **C++** and **Java**, are really a generalization of while loops. Consider the AST for a for loop, as shown in Figure 10.6.

As was the case the while loops, the for loop's AST contains subtrees corresponding to the loop's termination condition (**condition**) and its body (**loopBody**). In addition it contains ASTs corresponding to the loop's initialization (**initializer**) and its end-of-loop increment (**increment**).

There are a few differences that must be properly handled. Unlike the while loop, the for loop's termination condition is optional. (This allows "do forever" loops of the form `for (; ;) { ... }`). In **C++** and **Java**, an index local to the for loop may be declared, so a new symbol table name scope must be opened, and then closed, during semantic analysis.

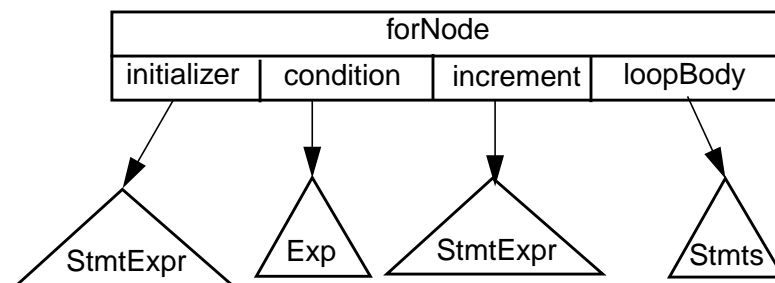


Figure 10.6 Abstract Syntax Tree for a For Loop

Reachability analysis is similar to that performed for while loops. The loop initializer is a declaration or statement expression; the loop increment is a statement expression; the termination condition is an expression. All of these are marked as terminating normally.

A null termination condition, or a constant expression equal to true, represent a non-terminating loop. The for loop is marked as not terminating normally (though a break within the loop body may change this when it is analyzed). A termination condition that is a constant expression equal to false causes the loop body to be marked as unreachable. If the control expression is non-null and non-constant, the loop is marked as terminating normally.

The `Semantics` method for a for loop is shown in Figure 10.7.

A new name scope is opened in case a loop index is declared in the initializer AST. The initializer is checked, allowing local index declarations to be processed. The increment is checked next. If condition is not a `NullNode`, it is type-checked, with a `boolean` (or `errorType`) required. The special cases of a null condition and a constant-valued condition are considered, with normal termination and reachability of the loop body updated appropriately. The `loopBody` AST is then checked. Finally, the name scope associated with for loop is closed and the loop's `throwsSet` is computed.

```

forNode.Semantics( )
1.  OpenNewScope()
2.  initializer.Semantics()
3.  increment.Semantics()
4.  terminatesNormally ← true
5.  loopBody.isReachable ← true
6.  if condition ≠ NullNode
7.      then ConditionType ← condition.ExprSemantics()
8.          if ConditionType = boolean
9.              then ConditionValue ← condition.EvaluateConstExpr()
10.                 if ConditionValue = true
11.                     then terminatesNormally ← false
12.                 elsif ConditionValue = false
13.                     then loopBody.isReachable ← false
14.             elsif ConditionType ≠ errorType
15.                 then GenerateErrorMessage("Illegal termination expression")
16.             else terminatesNormally ← false
17.  loopBody.Semantics()
18.  Close Scope()
19.  throwsSet ← union(initializer.throwsSet, condition.throwsSet,
                       increment.throwsSet, loopBody.throwsSet)

```

Figure 10.7 Semantic Checks a for For Loop

As an example, consider the following for loop

```

for (int i=0; i < 10; i++)
    a[i] = 0;

```

First a new name scope is created for the loop. When the declaration of `i` in the initializer AST is processed, it is placed in this new scope (since all new declarations are placed in the innermost open scope). Thus the references to `i` in the con-

dition, `increment` and `loopBody` ASTs properly reference the newly declared loop index `i`. Since the loop termination condition is boolean-valued and con-constant, the loop is marked as terminating normally, and the loop body is considered reachable. At the end of semantic checking, the scope containing `i` is closed, guaranteeing that not subsequent references to the loop index will be allowed.

A number of languages, including Fortran, Pascal, Ada, Modula 2, and Modula 3, contain a more restrictive form of for loop. Typically, a variable is identified as the “loop index.” Initial and final index values are defined, and sometimes an increment value is specified. For example, in Pascal a for loop is of the form

```
for id := initialVal to finalVal do
    loopBody
```

The loop index, `id`, must already be declared and must be a scalar type (integer or enumeration). The `initialVal` and `finalVal` expressions must be semantically valid and have the same type as the loop index. Finally, the loop index may not be changed within the `loopBody`. This can be enforced by marking `id`'s declaration as “constant” or “read only” while `loopBody` is being analyzed.

10.1.4 Break, Continue, Return and Goto Statements

Java contains no goto statement. It does, however, include break and continue statements which are restricted forms of a goto, as well as a return statement. We'll consider the continue statement first.

Continue Statements. Like the continue statement found in C and C++, Java's continue statement attempts to “continue with” the next iteration of a while, do or for loop. That is, it transfers control to the bottom of a loop where the loop index is iterated (in a for loop) and the termination condition is evaluated and tested.

A continue may only appear within a loop; this must be verified during semantic analysis. Unlike C and C++ a loop label may be specified in a continue statement. An unlabeled continue references the innermost for, while or do loop in which it is contained. A labeled continue references the enclosing loop that has the corresponding label. Again, semantic analysis must verify that an enclosing loop with the proper label exists.

Any statement in Java may be labeled. As shown in Figure 10.8 we'll assume an AST node `labeledStmt` that contains a string-valued field `stmtLabel`. If the statement is labeled, `stmtLabel` contains the label in string form. If the statement is unlabeled, `stmtLabel` is null. `labeledStmt` also contains a field `stmt` that is the AST node representing the labeled statement.

In Java, C and C++ (and most other programming languages) labels are placed in a different name space than other identifiers. This just means that an identifier used as a label may also be used for other purposes (a variable name, a type name, a method name, etc.) without confusion. This is because labels are

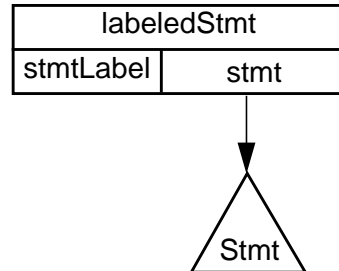


Figure 10.8 Abstract Syntax Tree for a Labeled Statement

used in very limited contexts (in continues, breaks and perhaps gotos). Labels can't be assigned to variables, returned by functions, read from files, etc.

We'll represent labels that are currently visible using a `labelList` variable. This variable is set to null when we begin the analysis of a method or constructor body, or a static initializer.

A `labelListNode` contains four fields: `label`, a string that contains the name of the label, `kind` (one of `iterative`, `switch` or `other`) that indicates the kind of statement that is labeled, `AST`, a link to the AST of the labeled statement and `next`, which is a link to the next `labelListNode` on the list.

Looking at a `labelList`, we can determine the statements that enclose a break or continue, as well as all the labels currently visible to the break or continue.

The semantic analysis necessary for a `labeledStmt` is shown in Figure 10.9. The `labelList` is extended by adding an entry for the current `labeledStmt` node, using its `label` (which may be null), and its `kind` (determined by a call to an auxiliary method, `getKind(stmt)`). The `stmt` AST is analyzed using the extended `labelList`. After analysis, `labelList` is returned to its original state, by removing its first element.

```

labeledStmt.Semantics ( )
1.  labelList ← labelListNode(stmtLabel, getKind(stmt), stmt, labelList)
2.  stmt.isReachable ← labeledStmt.isReachable
3.  stmt.Semantics()
4.  terminatesNormally ← stmt.terminatesNormally
5.  throwsSet ← stmt.throwsSet
6.  labelList ← labelList.next
  
```

Figure 10.9 Semantic Analysis for Labeled Statements

A continue statement without a label references the innermost iterative statement (while, do or for) within which it is nested. This is easily checked by looking for a node on the `labelList` with `kind = iterative` (ignoring the value of the `label` field).

A continue statement that references a label `L` (stored in AST field `stmtLabel`) must be enclosed by an iterative statement whose `label` is `L`. If more than one con-

taining statement is labeled with L, the nearest (innermost) is used. The details of this analysis are shown in Figure 10.10.

```

ContinueNode.Semantics ( )
1.  terminatesNormally ← false
2.  throwsSet ← null
3.  currentPos ← labelList
4.  if stmtLabel = null
5.      then while currentPos ≠ null
6.          do if currentPos.kind = iterative
7.              then return
8.              currentPos = currentPos.next
9.          GenerateErrorMessage("Continue not inside iterative statement")
10. else while currentPos ≠ null
11.     do if currentPos.label = stmtLabel and
12.         currentPos.kind = iterative
13.         then return
14.         currentPos = currentPos.next
15.     GenerateErrorMessage("Continue label doesn't match an
16.                             iterative statement")

```

Figure 10.10 Semantic Analysis for Continue Statements

As an example, consider the following code fragment

```

L1: while (p != null) {
    if (p.val < 0)
        continue
    else ... }

```

The `labelList` in use when the `continue` is analyzed is shown in Figure 10.11. Since the list contains a node with `kind = iterative`, the `continue` is correct.

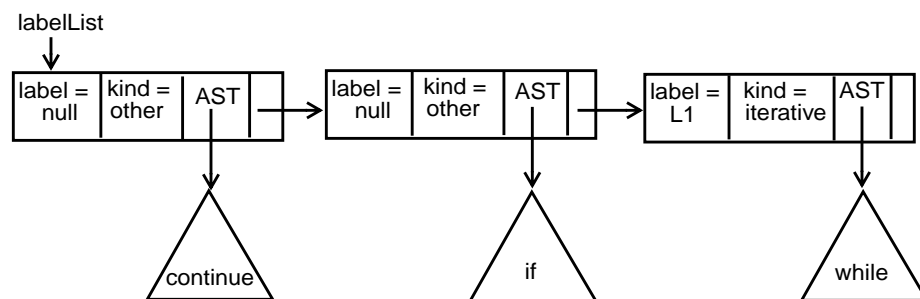


Figure 10.11 Example of a LabelList in a Continue Statement

In `C` and `C++` semantic analysis is even simpler. `Continues` don't use labels so the innermost iterative statement is always selected. This means we need only lines 1 to 9 of Figure 10.10.

Break Statements. In Java an unlabeled break statement has the same meaning as the break statement found in C and C++. The innermost while, do, for or switch statement is exited, and execution continues with the statement immediately following the exited statement. Thus a reachable break forces the statement it references to terminate normally.

A labeled break exits the enclosing statement with a matching label (not necessarily a while, do, for or switch statement), and continues execution with that statement's successor (again, if reachable, it forces normal termination of the labeled statement). For both labeled and unlabeled breaks, semantic analysis must verify that a suitable target statement for the break exists.

We'll again use the `labelList` introduced in the last section. For unlabeled breaks, we'll need to find a node with `kind = iterative` or `switch`. For labeled breaks, we'll need to find a node with a matching label (its `kind` doesn't matter). In either case, the `terminatesNormally` field of the referenced statement will be set to true if the break is marked as reachable.

This analysis is detailed in Figure 10.12.

```

BreakNode.Semantics ( )
1.  terminatesNormally ← false
2.  throwsSet ← null
3.  currentPos ← labelList
4.  if stmtLabel = null
5.      then while currentPos ≠ null
6.          do if currentPos.kind = iterative
              or currentPos.kind = switch
7.              then if isReachable
8.                  then currentPos.AST.terminatesNormally ← true
9.                  return
10.             currentPos = currentPos.next
11.         GenerateErrorMessage("Break not inside iterative or
                               switch statement")
12.     else while currentPos ≠ null
13.         do if currentPos.label = stmtLabel
14.             then if isReachable
15.                 then currentPos.AST.terminatesNormally ← true
16.                 return
17.             currentPos = currentPos.next
18.         GenerateErrorMessage("Continue label doesn't match any
                               statement label")

```

Figure 10.12 Semantic Analysis for Break Statements

As an example, consider the following code fragment

```

L1: for (i=0; i < 100; i++)
    for (j=0; j < 100; j++)

```

```

if (a[i][j] == 0)
    break L1;
else ...
    
```

The `labelList` in use when the `break` is analyzed is shown in Figure 10.13. Since the list contains a node with `label = L1`, the `break` is correct. The `for` loop labeled with `L1` is marked as terminating normally.

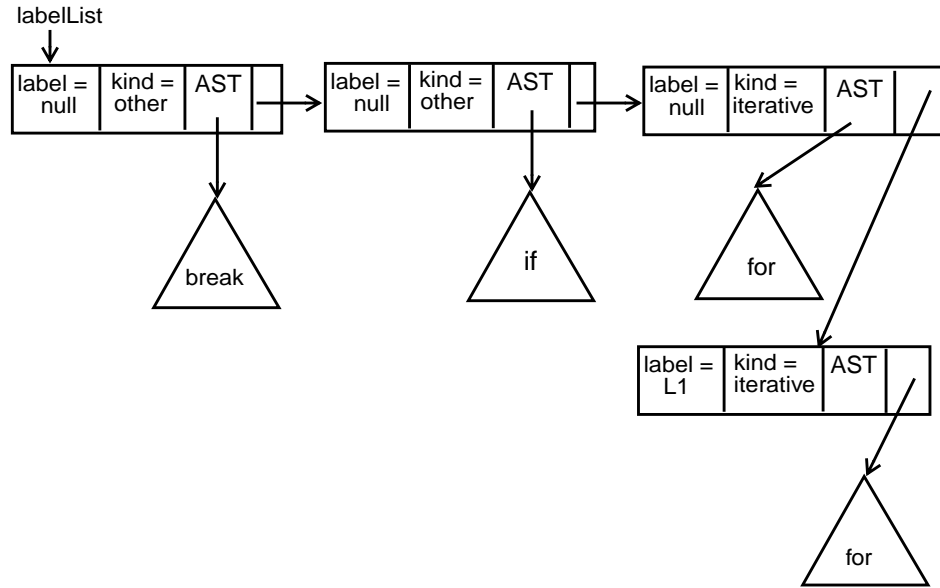


Figure 10.13 Example of a LabelList in a Break Statement

Return Statements. An AST rooted by a `returnNode`, as shown in Figure 10.14, represents a return statement. The field `returnVal` is null if no value is returned; otherwise it is an AST representing an expression to be evaluated and returned.

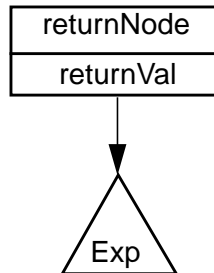


Figure 10.14 Abstract Syntax Tree for a Return Statement

The semantic rules governing a return statement depend on where the statement appears. A return statement without an expression value may only appear in a void method (a subroutine) or in a constructor. A return statement with a return value may only appear in a method whose type may be assigned the return type (this excludes void methods and constructors). A return (of either form) may not appear in a static constructor.

A method declared to return a value (a function) **must** exit via a return of a value or by throwing an exception. This requirement can be enforced by verifying that the statement list that comprises a function's body has its `terminatesNormally` value set to false.

To determine the validity of a return we will check the kind of construct (method, constructor or static initializer) within which it appears. But AST links all point downward; hence looking "upward" is difficult. To assist our analysis, we'll assume three global pointers, `currentMethod`, `currentConstructor` and `currentStaticInitializer`, set during semantic analysis.

If we are checking an AST node contained within the body of a method, `currentMethod` will tell us which one. If we are analyzing an AST node not within a method, then `currentMethod` is null. The same is true for `currentConstructor` and `currentStaticInitializer`. We can determine which kind of construct we are in (and details of its declaration) by using the pointer that is non-null.

The details of semantic analysis for return statements appears in Figure 10.15. For methods we assume `currentMethod.returnType` is the type returned by the method (possibly equal to void). The auxiliary method is `assignable(T1,T2)` tests whether type T2 is assignable to type T1 (using the assignability rules of the languages being compiled).

```

return.Semantics( )
1.  terminatesNormally ← false
2.  if currentStaticInitializer ≠ null
3.      then GenerateErrorMessage("A return may not appear within a
                                     static initializer")
4.  elsif returnVal ≠ null
5.      then returnVal.ExprSemantics()
6.          throwsSet ← returnVal.throwsSet
7.          if currentMethod = null
8.              then GenerateErrorMessage("A value may not be returned
                                             from a constructor")
9.          elsif not assignable(currentMethod.returnType,returnVal.type)
10.             then GenerateErrorMessage("Illegal return type")
11.         else if currentMethod ≠ null and currentMethod.returnType ≠ void
12.             then GenerateErrorMessage("A value must be returned")
13.         throwsSet ← null

```

Figure 10.15 Semantic Analysis for Return Statements

C++ has semantic rules very similar to those of **Java**. A value may only be returned from a non-void function, and the value returned must be assignable to

the function's return type. In **C** a return without a value is allowed in a non-void function (with undefined behavior).

Goto Statements. **Java** contains no goto statement, but many other languages, including **C** and **C++**, do. **C**, **C++** and most other languages that allow gotos restrict them to be **intraprocedural**. That is, a label and all gotos that reference it must be in the same procedure or function.

As noted earlier, identifiers used as labels are usually considered distinct from identifiers used for other purposes. Thus in **C** and **C++**, the statement

```
a: a+1;
```

is legal. Labels may be kept in a separate symbol table, distinct from the main symbol table used to store ordinary declarations.

Labels need not be defined before they are used; “forward gotos” are allowed. Semantic checking must guarantee that all labels used in gotos are in fact defined somewhere in the current procedure.

Because of potential forward references, it is a good idea to check labels and gotos in two steps. First, the AST that represents the entire body of a subprogram is traversed, gathering all label declarations into a **declaredLabels** table stored as part of the current subprogram's symbol table information. Duplicate labels are detected as **declaredLabels** is built.

During normal semantic processing of the body of a subprogram (after **declaredLabels** has been built), an AST for a goto can access **declaredLabels** (through the current subprogram's symbol table). Checking for valid label references (whether forward or not) is easy.

A few languages, like **Pascal**, allow **non-local gotos**. A non-local goto transfers control to a label in a scope that contains the current procedure. Non-local gotos can be checked by maintaining a stack (or list) of **declaredLabels** tables, one for each nested procedure. A goto is valid if its target appears in any of the **declaredLabels** tables.

Finally, some programming languages forbid gotos into a conditional or iterative statement from outside. That is, even if the scope of a label is an entire subprogram, a goto into a loop or from a then part to an else part is forbidden. Such restrictions can be enforced by marking each label in **declaredLabels** as either “active” or “inactive.” Gotos are allowed only to active labels, and a label within a conditional or iterative statement is active only while the AST that contains the label is being processed. Thus a label **L** within a while loop becomes active when the loop body's AST is checked, and is inactive when statements outside the loop body are checked.

10.1.5 Switch and Case Statements

Java, **C** and **C++** contain a **switch** statement that allows the selection of one of a number of statements based on the value of a control expression. **Pascal**, **Ada** and

Modula 3 contain a **case** statement that is equivalent. We shall focus on translating switch statements, but our discussion applies equally to case statements.

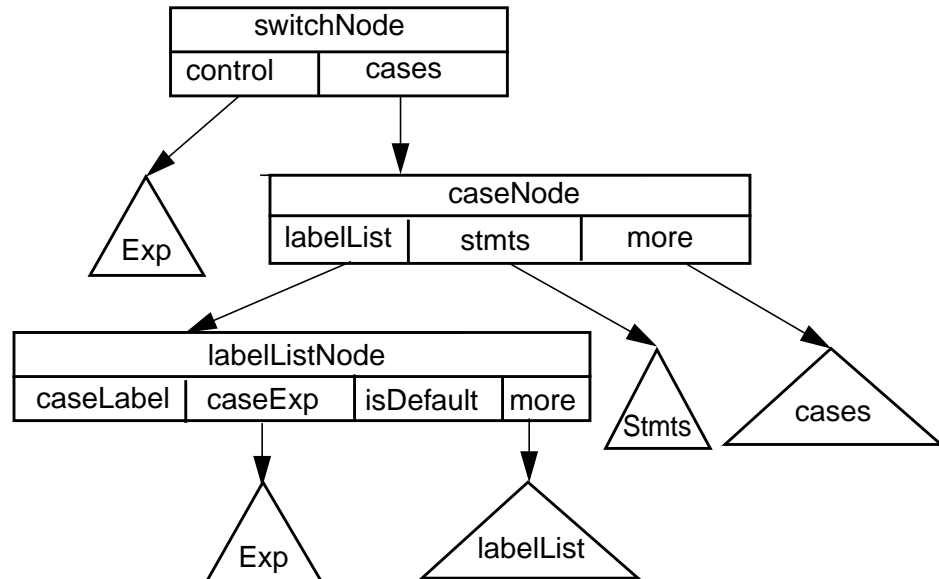


Figure 10.16 Abstract Syntax Tree for a Switch Statement

The AST for a switch statement, rooted at a **switchNode** is shown in Figure 10.16 (fields not needed for semantic analysis are omitted for clarity). In the AST **control** represents an integer-valued expression; **cases** is a **caseNode**, representing the cases in the switch. Each **caseNode** has three fields. **labelList** is a **labelListNode** that represents one or more case labels. **stmts** is an AST node representing the statements following a case constant in the switch. **more** is either null or another **caseListNode**, representing the remaining cases in the switch.

A **labelListNode** contains an integer field **caseLabel**, an AST **caseExp**, a boolean field **isDefault** (representing the default case label) and **more**, a field that is either null or another **labelListNode**, representing the remainder of the list. The **caseExp** AST represents a constant expression that labels a case within the switch; when it is evaluated, **caseLabel** will hold its value.

A number of steps are needed to check the semantic correctness of a switch statement. The control expression and all the statements in the case body must be checked. The control expression must be an integer type (32 bits or less in size). Each case label must be a constant expression assignable to the control expression. No two case labels may have the same value. At most one default label may appear within the switch body.

A switch statement can terminate normally in a number of ways. An empty switch body (uncommon, but legal) trivially terminates normally. If the last switch group (case labels followed by a statement list) terminates normally, so does the

switch (since execution “falls through” to the succeeding statement. If any of the statements within a switch body contain a reachable break statement, then the entire switch can terminate normally.

We’ll need a number of utility routines in our semantic analysis. To process case labels, we will build a linked list of `node` objects. Each `node` will contain a `value` field (an integer) and a `next` field (a reference to a node).

The routine `buildLabels` (Figure 10.17) will traverse a `labelListNode` (including its successors), checking the validity of case labels and building a list of `node` objects representing the case labels that are encountered. The related routine `buildLabelList` (Figure 10.17) will traverse a `caseNode` (including its successors). It checks the validity of statements within each `caseNode` and builds a list of `node` objects representing the case labels that are found within the `caseNode`.

```

buildLabels( labelList )
1.  if labelList = null
2.    then return null
3.  elsif isDefault
4.    then return buildLabels(more)
5.    else caseExp.ExprSemantics()
6.         if caseExp.type = errorType
7.           then return buildLabels(more)
8.         elsif not assignable(CurrentSwitchType, CaseExp.type)
9.           then GenerateErrorMessage("Invalid Case Label Type")
10.            return buildLabels(more)
11.         else caseLabel ← caseExp.EvaluateConstExpr()
12.            if caseLabel = null
13.              then GenerateErrorMessage("Case Label Must be
14.                a Constant Expression")
15.              return buildLabels(more)
16.            else return node(caseLabel, buildLabels(more))

buildLabelList ( cases )
1.  stmts.isReachable ← true
2.  stmts.Semantics()
3.  if more = null
4.    then terminatesNormally ← stmts.terminatesNormally
5.         throwsSet ← stmts.throwsSet
6.         return buildLabels(labelList)
7.    else restOfLabels ← buildLabelList(more)
8.         terminatesNormally ← more.terminatesNormally
9.         throwsSet ← union(stmts.throwsSet, more.throwsSet)
10.     return append(buildLabels(labelList), restOfLabels)

```

Figure 10.17 Utility Semantic Routines for Switch Statements (Part 1)

The routine `checkForDuplicates` (Figure 10.18) takes a sorted list of `node` objects, representing all the labels in the switch statement, and checks for duplicates by comparing adjacent values. Routines `countDefaults` and `countDefaultLabels` (Figure 10.18) count the total number of default cases that appear within a `switchNode`.

```

checkForDuplicates( node )
1.  if node ≠ null and node.next ≠ null
2.      then if node.value = node.next.value
3.          then GenerateErrorMessage("Duplicate case label:",
                                     node.value)
4.          checkForDuplicates(node.next)

countDefaults( cases )
1.  if cases = null
2.      then return 0
3.      else return countDefaultLabels(labelList) + countDefaults(more)

countDefaultLabels( labelList )
1.  if labelList = null
2.      then return 0
3.      if isDefault
4.          then return 1 + countDefaultLabels(more)
5.          else return countDefaultLabels(more)

```

Figure 10.18 Utility Semantic Routines for Switch Statements (Part 2)

We can now complete the definition of semantic processing for switch statements, as shown in Figure 10.19. We first mark the whole switch as not terminating normally. This will be updated to true if `cases` is null, or if a reachable break is encountered while checking the switch body or if the `stmts` AST of the last `caseNode` in the AST is marked as terminating normally. The switch's control expression is checked. It must be a legal expression and assignable to type `int`. A list of all the case label values is built by traversing the `cases` AST. As the label list is built, case statements and case labels are checked. After the label list is built, it is sorted. Duplicates are found by comparing adjacent values in the sorted list. Finally, the number of default cases is checked by traversing the `cases` AST again.

As an example, consider the following switch statement

```

switch(p) {
  case 2:
  case 3:
  case 5:
  case 7: isPrime = true; break;
  case 4:
  case 6:

```

```

switchNode.Semantics( )
1.  terminatesNormally ← false
2.  control.Semantics()
3.  if control.type ≠ errorType and not assignable(int, control.type)
4.      then GenerateErrorMessage("Illegal Type for Control Expression")
5.  CurrentSwitchType ← control.type
6.  if cases = null
7.      then terminatesNormally ← true
8.          throwsSet ← control.throwsSet
9.      else labelList ← buildLabelList(cases)
10.     terminatesNormally ← terminatesNormally
        or cases.terminatesNormally
11.     throwsSet ← union(control.throwsSet, cases.throwsSet)
12.     labelList ← sort(labelList)
13.     checkForDuplicates(labelList)
14.     if countDefaults(cases) > 1
15.         then GenerateErrorMessage("More than One Default
                Case Label")

```

Figure 10.19 Semantic Analysis for Switch Statements

```

case 8:
case 9:isPrime = false; break;
default:isPrime = checkIfPrime(p);
}

```

Assume that p is declared as an integer variable. We check p and find it a valid control expression. The label list is built by examining each `caseNode` and `labelListNode` in the AST. In doing so, we verify that each case label is a valid constant expression that is assignable to p . The case statements are checked and found valid. Since the last statement in the switch (the default) terminates normally, so does the entire switch statement. The value of `labelList` is $\{2,3,5,7,4,6,8,9\}$. After sorting we have $\{2,3,4,5,6,7,8,9\}$. No adjacent elements in the sorted list are equal. Finally, we count the number of default labels; a count of 1 is valid.

The semantic rules for switch statements in **C** and **C++** are almost identical to those of **Java**.

Other languages include a case statement that is similar in structure to the switch statement. **Pascal** allows enumerations as well as integers in case statements. It has no default label and considers the semantics of an unmatched case value to be undefined. Some **Pascal** compilers check for complete case coverage by comparing the sorted label list against of range of values possible in the control expression (**Pascal** includes subrange types that limit the range of possible values a variable may hold).

Ada goes farther than **Pascal** in requiring that all possible control values must be covered within a case statement (though it does allow a default case). Again, comparing sorted case labels against possible control expression values is required.

Ada also generalizes a case label to a **range** of case values (e.g., in Java notation, case 1..10, which denotes 10 distinct case values). Semantic checks that look for duplicate case values and check for complete coverage of possible control values must be generalized to handle ranges rather than singleton values.

10.1.6 Exception Handling

Java, like most other modern programming languages, provides an **exception handling** mechanism. During execution, an exception may be **thrown**, either explicitly (via a throw statement) or implicitly (due to an execution error). Thrown exceptions may be **caught** by an **exception handler**.

Exceptions form a clean and general mechanism for identifying and handling unexpected or erroneous situations. They are clearer and more efficient than using error flags or gotos. Though we will focus on Java's exception handling mechanism, most recent language designs, including C++, Ada and ML, include an exception handling mechanism similar to that of Java.

Java exceptions are **typed**. An exception throws an object that is an instance of class `Throwable` or one of its subclasses. The object thrown may contain fields that characterize the precise nature of the problem the exception represents, or the class may be empty (with its type signifying all necessary information).

Java exceptions are classified as either **checked** or **unchecked**. A checked exception thrown in a statement must be caught in an enclosing try statement or listed in the throws list of the enclosing method or constructor. Thus it must be handled or listed as a possible result of executing a method or constructor.

An unchecked exception (defined as an object assignable to either class `RuntimeException` or class `Error`) may be handled in a try statement, but need not be. If uncaught, unchecked exceptions will terminate execution. Unchecked exceptions represent errors that may appear almost anywhere (like accessing a null reference or using an illegal array index). These exceptions usually force termination, so explicit handlers may clutter a program without adding any benefit (termination is the default for uncaught exceptions).

We'll first consider the semantic checking needed for a try statement, whose AST is shown in Figure 10.20.

To begin we will check the optional finally clause (referenced by `final`). The correctness of statements within it is independent of the contents of the try block and the catch clauses.

Next, the catch clauses must be checked. Each catch clause is represented by a `catchNode`, as shown in Figure 10.21.

Catch clauses require careful checking. Each clause introduces a new identifier, the parameter of the clause. This identifier must be declared as an exception (of class `Throwable` or a subclass of it). The parameter is local to the body of the catch clause, and must not hide an already visible local variable or parameter.

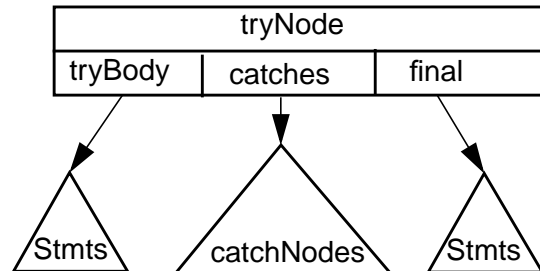


Figure 10.20 Abstract Syntax Tree for a Try Statement

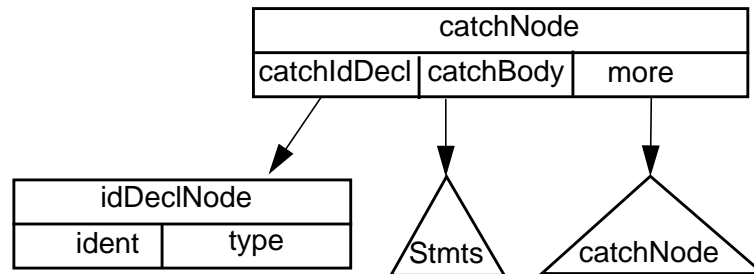


Figure 10.21 Abstract Syntax Tree for a Catch Block

Moreover, within the catch body, the catch parameter may not be hidden by a local declaration.

We compute the exceptions that are caught by the catch clauses (**localCatches**) and verify (using method `isSubsumedBy`, defined in Figure 10.22) that no catch clause is “hidden” by earlier catches. This is a reachability issue—some exception type must be able to reach, and activate, each of the catch clauses.

Finally, the body of the try block is checked. Statements within the try body must know what exceptions will be caught by the try statement. We will create a data structure called **catchList** that is a list of **throwItem** nodes. Each **throwItem** on the list represents one exception type named in a catch block of some enclosing try statement.

Before we begin the semantic analysis of the current **tryNode**, the **catchList** contains nodes for all the exceptions caught by try statements that enclose the current try statement (initially this list is null).

We compute the exceptions handled by the current catch clauses, building a list called **localCatches**. We extend the current **catchList** by prepending **localCatches**. The extended list is used when the try statements are checked, guaranteeing that the definitions of local exception handlers are made visible within the try body.

After the try body is analyzed, we compute the `throwsSet` for the `tryNode`. This is the set of exceptions that “escape” from the try statement. It is the set of exceptions thrown in the finally clause, or any catch clause, plus exceptions generated in the try body, but not caught in an of the catch clauses. The method `filterOutThrows`, defined in Figure 10.22, details this process.

The complete semantic processing for a `tryNode` is detailed in Figure 10.23.

```
isSubsumedBy( catchList, exceptionType )
```

1. while `catchList` \neq null
2. do if `assignable(catchList.type, exceptionType)`
3. then return true
4. else `currentList` \leftarrow `currentList.next`
5. return false

```
filterOutThrows( throwsList, exceptionType )
```

1. if `throwsList` = null
2. then return null
3. elsif `assignable(exceptionType, throwsList.type)`
4. then return `filterOutThrows(throwsList.next, exceptionType)`
5. else return `throwItem(throwsList.type,`
`filterOutThrows(throwsList.next, exceptionType)`)

Figure 10.22 Utility Routines for Catch Clauses.

The AST for a throw statement is shown in Figure 10.24. The type of value thrown in a throw statement must, of course, be an exception (a type assignable to `Throwable`). If the exception thrown is checked, then semantic analysis must also verify that an enclosing try block can catch the exception or that the method or constructor that contains the throw has included the exception in its throws list.

The set of exceptions generated by the throws statement (its `throwsSet`) is the exception computed by the `thrownVal` AST **plus** any exceptions that might be thrown by the expression that computes the exception value.

In processing statements we built a `throwsSet` that represents all exception types that may be thrown by a statement. We will assume that when the header of a method or constructor is semantically analyzed, we create a `declaredThrowsList` (composed of `throwItem` nodes) that represents all the exception types (if any) mentioned in the throws list of the method or constructor. (A static initializer will always have a null `declaredThrowsList`). Comparing the `throwsSet` and `declaredThrowsList`, we can readily check whether all checked exceptions are properly handled.

Semantic analysis of a throw statement is detailed in Figure 10.25.

As an example, consider the following Java code fragment

```
class ExitComputation extends Exception{};
try { ...
```



```

tryNode.Semantics( )
1.  terminatesNormally ← false
2.  tryBody.isReachable ← final.isReachable ← true
3.  final.Semantics()
4.  currentCatch ← catches
5.  localCatches ← throwsInCatches ← null
6.  while currentCatch ≠ null
7.      do if not assignable(Throwable, currentCatch.catchIdDecl.type)
8.          then GenerateErrorMessage("Illegal type for catch identifier")
9.              currentCatch.catchIdDecl.type ← errorType
10.         elsif isSubsumedBy(localCatches, currentCatch.catchIdDecl.type)
11.             then GenerateErrorMessage("Catch is Hidden by Earlier Catches")
12.             else localCatches ← append(localCatches,
13.                                     catchNode(currentCatch.catchIdDecl.type,null)
14.             currentDecl ← Lookup(currentCatch.catchIdDecl.ident)
15.             if currentDecl ≠ null and
16.                 (currentDecl.kind = Variable or currentDecl.kind = Parameter)
17.                 then GenerateErrorMessage("Attempt to redeclare local identifier")
18.             OpenNewScope()
19.             DeclareLocalVariable(currentCatch.catchIdDecl.ident,
20.                                 currentCatch.catchIdDecl.type, CantBeHidden)
21.             catchBody.isReachable ← true
22.             currentCatch.catchBody.Semantics()
23.             terminatesNormally ← terminatesNormally or
24.                                 currentCatch.catchBody.terminatesNormally
25.             throwsInCatch ←
26.                 union(throwsInCatch, currentCatch.catchBody.throwsSet)
27.             CloseScope()
28.             currentCatch ← currentCatch.more
29.         prevCatchList ← catchList
30.         catchList ← append(localCatches, catchList)
31.         tryBody.Semantics()
32.         terminatesNormally ← (terminatesNormally or tryBody.terminatesNormally)
33.                             and final.terminatesNormally
34.         catchList ← prevCatchList
35.         throwsInTry ← tryBody.throwsSet
36.         currentCatch ← catches
37.         while currentCatch ≠ null
38.             do newSet ← filterOutThrows(throwsInTry, currentCatch.catchIdDecl.type)
39.             if newSet = throwsInTry
40.                 then GenerateErrorMessage("No Throws Reach this Catch")
41.             else throwsInTry ← newSet
42.             currentCatch ← currentCatch.more
43.         throwsSet ← union(throwsInTry, throwsInCatch, final.throwsSet)

```

Figure 10.23 Semantic Analysis for Throw Statements

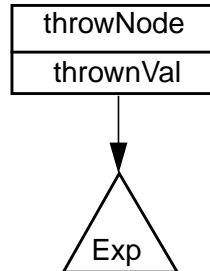


Figure 10.24 Abstract Syntax Tree for a Throw Statement

```

throwNode.Semantics( )
1.  terminatesNormally ← false
2.  throwsSet ← null
3.  thrownType ← thrownVal.ExprSemantics()
4.  if thrownType ≠ errorType
5.    then if not assignable(Throwable, thrownType)
6.          then GenerateErrorMessage("Illegal type for throw")
7.          else if assignable(RuntimeException, thrownType) or
8.                assignable(Error, thrownType)
9.                then return
10.             throwsSet ← union(thrownVal.throwsSet,
11.                               throwItem(thrownType, null))
12.             throwTargets ← append(catchList, declaredThrowsList)
13.             while throwTargets ≠ null
14.               do if assignable(throwTargets.type, thrownType)
15.                   then return
16.                   else thrownType ← thrownType.next
17.             GenerateErrorMessage("Type thrown not found in
18.                                   enclosing catch or declared throws list")
  
```

Figure 10.25 Semantic Analysis for Throw Statements

```

if (cond)
    throw new ExitComputation();
if (v < 0.0)
    throw new ArithmeticException();
else a = Math.sqrt(v);
... }
catch (e ExitComputation) {return 0;}
  
```

A new checked exception, `ExitComputation`, is declared. In the try statement, we first check the catch clause. Assuming `e` is not already defined as a variable or parameter, no errors are found. The current `catchList` is extended with an

entry for type `ExitComputation`. The try body is then checked. Focusing on throw statements, we first process a throw of an `ExitComputation` object. This is a valid subclass of `Throwable` and `ExitComputation` is on the `catchList`, so no errors are detected. Next the throw of an `ArithmeticException` is checked. It too is a valid exception type. It is an unchecked exception (a subclass of `RuntimeException`), so the throw is valid independent of any try statements that enclose it.

The exception mechanism of `C++` is very similar to that of `Java`, using an almost identical throw/catch mechanism. The techniques developed in this section are directly applicable.

Other languages, like `Ada`, feature a single exception type that is “raised” rather than thrown. Exceptions are handled in a “when” clause that can be appended to any begin-end block. Again, semantic processing is very similar to the mechanisms developed here.

10.2 Semantic Analysis of Calls

In this section we investigate the semantic analysis of method calls in `Java`. The techniques we present are also applicable to constructor and interface calls, as well as calls to subprograms in `C`, `C++` and other languages.

The AST for a `callNode` is shown in Figure 10.26. The field `method` is an identifier that specifies the name of the method to be called. The field `qualifier` is an optional expression that specifies the object or class within which method is to be found. Finally, `args` is an optional expression list that represents the actual parameters to the call.

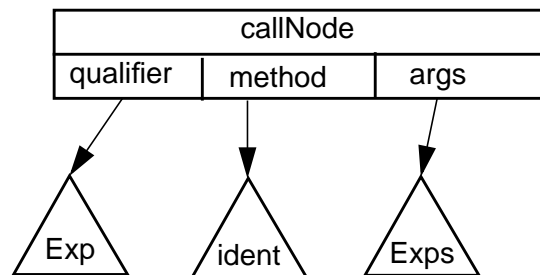


Figure 10.26 Abstract Syntax Tree for a Method Call

The first step in analyzing a call is to determine which method definition to use. This determination is by no means trivial in `Java` because of **inheritance** and **overloading**.

Recall that classes form an inheritance hierarchy, and all classes are derived, directly or indirectly, from `Object`. An object may have access to methods defined

in its own class, its parent class, its grandparent class, and so on, all the way up to `Object`. In processing a call all potential locales of definition must be checked.

Because of overloading, it is valid to define more than one method with the same name. A call must select the “right” definition, which informally is the nearest accessible method definition in the inheritance hierarchy whose parameters match the actual parameters provided in the call.

Let us begin by gathering all the method definitions that might be applicable to the current call. This lookup process is guided by the kind of method qualifier (if any) that is provided and the access mode of individual methods:

- If no qualifier is provided, we examine the class (call it `C`) that contains the call being analyzed. All methods defined within `C` are accessible. In addition, as explained in section xxx.yyy, methods defined in `C`’s superclasses (its parent class, grandparent class, etc.) may be inherited depending on their access qualifiers:
 - ❑ Methods marked `public` or `protected` are always included.
 - ❑ A method with default access (not marked `public`, `private` or `protected`) is included if the class within which it is defined is in the same package as `C`.
 - ❑ Private methods are **never** included (since they can’t be inherited)
- If the qualifier is the reserved word `super`, then a call of `M` in class `C` must reference a method inherited from a superclass (as defined above). Use of `super` in class `Object` is illegal, because `Object` has no superclass.
- If the qualifier is a type name `T` (which must be a class name), then `T` must be in the package currently being compiled, or it must be a class marked `public`. A call of `M` must reference a static method (instance methods are disallowed because the object reference symbol `this` is undefined). The static methods that may be referenced are:
 - ❑ All public methods defined in `T` or a superclass of `T`.
 - ❑ Methods defined in `T` or a superclass with default access if the defining class occurs within the current package.
 - ❑ Methods defined in `T` or a superclass of `T` marked `protected` if they are defined in the current package or if the call being analyzed occurs within a subclass of `T`.
- If the qualifier is an expression that computes an object of type `T`, then `T` must be in the package currently being compiled, or it must be a class marked `public`. A call of `M` may reference:
 - ❑ All public methods defined in `T` or a superclass of `T`.
 - ❑ Methods defined in `T` or a superclass with default access if the class containing the method definition occurs within the current package.

- Methods defined in T or a superclass of T marked protected if they are defined in the current package or if T is a subclass of C , the class that contains the call being analyzed.

These rules for selecting possible method definitions are codified in Figure 10.27. We assume that `methodDefs(ID)` returns all the methods named `ID` in a given class. Similarly, `publicMethods(ID)` returns all the public methods named `ID`, etc. The reference `currentClass` accesses the class currently being compiled; `currentPackage` references the package being currently compiled.

```

callNode.getMethods ( )
1.  if qualifier = null
2.    then methodSet ← currentClass.methodDefs(method)
3.    else methodSet ← null
4.  if qualifier = null or qualifier = superNode
5.    then nextClass ← currentClass.parent
6.    else nextClass ← qualifier.type
7.  while nextClass ≠ null
8.    do if qualifier ≠ null and qualifier ≠ superNode and
          nextClass.package ≠ currentPackage and not nextClass.isPublic
9.        then nextClass ← nextClass.parent
10.       continue
11.     methodSet ← union(methodSet, nextClass.publicMethods(method))
12.     if nextClass.package = currentPackage
13.       then methodSet ← union(methodSet,
                                nextClass.defaultAccessMethods(method))
14.     if qualifier = null or qualifier = superNode
          or nextClass.package = currentPackage
          or (qualifier.kind = type and isAncestor(qualifier.type, currentClass)
          or (qualifier.kind = value and isAncestor(currentClass, qualifier.type))
15.       then methodSet ← union(methodSet,
                                nextClass.protectedMethods(method))
16.     nextClass ← nextClass.parent
17.  return methodSet

```

Figure 10.27 Compute Set of Method Definitions that Match the Current Call

Once we have determined the set of definitions that are *possible*, we must filter them by comparing these definitions with the number and type of expressions that form the call's actual parameters. Assume that an expression list is represented by an `exprsNode` AST (Figure 10.28) and is analyzed using the `Semantics` method defined in Figure 10.29.

We will assume that each method definition included in the set of accessible methods is represented as a `methodDefItem`, which contains the fields `returnType`, `argTypes`, and `classDefIn`. `returnType` is the type returned by the method; `classDefIn` is the class the method is defined in; `argTypes` is a linked list of `typeItem` nodes, one for each declared parameter. Each `typeItem` contains a `type` field (the

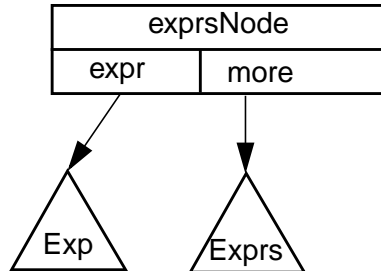


Figure 10.28 Abstract Syntax Tree for an Expression List

```

exprsNode.Semantics( )
1.  expr.ExprSemantics()
2.  more.Semantics()
3.  throwsSet ← union(expr.throwsSet, more.throwsSet)
  
```

Figure 10.29 Semantic Checks for an Expression List

type of the parameter) and `next`, (a reference to the next `typeItem` on the list). We can build a `typeItem` list for the actual parameters of the call using the `getArgTypes` method defined in Figure 10.30.

```

getArgTypes( exprList )
1.  if exprList = null
2.    then return null
3.    else return typeItem(exprList.expr.type, getArgTypes(exprList.more))
  
```

Figure 10.30 Build a Type List for an Expression List.

Once we have a type list for the actual parameters of a call, we can compare it with the declared parameter type list of each method returned by `findDefs`. But what exactly defines a match between formal and actual parameters? First, both argument lists must have the same length—this is easy to check. Next, each actual parameter must be “bindable” to its corresponding formal parameter.

Bindable means that it is legal to use an actual parameter whenever the corresponding formal parameter is referenced. In **Java**, `bindable` is almost the same as `assignable`. The only difference is that an integer literal, used as an actual parameter, may not be bound to a formal of type `byte`, `short` or `char`. (This requirement was added to make it easier to select among overloaded definitions). We will use the predicate `bindable(T1,T2)` to determine if an actual of type `T2` may be bound to a formal of type `T1`.

Now checking the feasibility of using a particular method definition in a call is straightforward—we check that the number of parameters is correct and that each parameter is `bindable`. This is detailed in Figure 10.31, which defines the method `applicable(formalParms,actualParms)`. If `applicable` returns true, a particular method definition can be used; otherwise, it is immediately rejected as not applicable to the call being processed.

```

applicable( formalParms, actualParms )
1.  if formalParms = null and actualParms = null
2.      then return true
3.  elsif formalParms = null or actualParms = null
4.      then return false
5.  elsif bindable(formalParms.type, actualParms.type)
6.      then return applicable(formalParms.next, actualParms.next)
7.      else return false

```

Figure 10.31 Test if Actual Parameters are Bindable to Corresponding Actual Parameters.

When analyzing a method call we will first select the set of method definitions the call might reference. Next we analyze the actual parameters of the call, and build an actual parameters list. This list is compared with the formal parameters list of each method under consideration, filtering out those that are not applicable (because of an incorrect argument count or an argument type mismatch).

At this stage we count the number of method definitions still under consideration. If it is zero, we have an invalid call—no accessible method can be called without error. If the count is one, we have a correct call.

If two or more method definitions are still under consideration, we need to choose the most appropriate definition. Two issues are involved here. First, if a method is redefined in a subclass, we want to use the redefinition. For example, method `M()` is defined in both classes `C` and `D`:

```

class C { void M() { ... } }
class D extends C { void M() { ... } }

```

If we call `M()` in an instance of class `D`, we want to use the definition of `M` in `D`, even though `C`'s definition is visible and type-correct.

It may also happen that one definition of a method `M` takes an object of class `A` as a parameter, whereas another definition of `M` takes a subclass of `A` as a parameter. An example of this is

```

class A { void M(A parm) { ... } }
class B extends A { void M(B parm) { ... } }

```

Now consider a call `M(b)` in class `B`, where `b` is of type `B`. Both definitions of `M` are possible, since an object of class `B` may always be used where a parameter of its parent class (`A`) is expected.

In this case we prefer to use the definition of `M(B parm)` in class `B` because it is a “closer match” to the call `M(b)` this is being analyzed.

We formalize the notion of one method definition being a “closer match” than another by defining one method definition `D` to be more specific than another definition `E` if `D`'s class is bindable to `E`'s class and each of `D`'s parameters is bindable to the corresponding parameter of `E`. This definition captures the notion that we prefer a method definition in a subclass to an otherwise identical definition in a parent class (a subclass may be assigned to a parent class, but not vice-versa). Similarly, we prefer arguments that involve a subclass over an argument that involves a parent class (as was the case in the example of `M(A parm)` and `M(B parms)` used above).

A method `moreSpecific(Def1,Def2)` that tests whether method definition `Def2` is more specific than method definition `Def1` is presented in Figure 10.32.

```

moreSpecific( Def1, Def2 )
1.  if bindable(Def1.classDefIn, Def2.classDefIn)
2.      then arg1 ← Def1.argTypes
3.          arg2 ← Def2.argTypes
4.          while arg1 ≠ null
5.              do if bindable(arg1.type, arg2.type)
6.                  then arg1 ← arg1.next
7.                      arg2 ← arg2.next
8.                  else return false
9.      return true
10. else return false

```

Figure 10.32 Test if One Method Definition is More Specific than Another Definition.

Whenever we have more than one accessible method definition that matches a particular argument list in a call, we will filter out less specific definitions. If, after filtering, only one definition remains (called the **maximally specific** definition), we know it is the correct definition to use. Otherwise, the choice of definition is **ambiguous** and we must issue an error message.

The process of filtering out less specific method definitions is detailed in Figure 10.33, which defines the method `filterDefs(MethodDefSet)`.

```

filterDefs( MethodDefSet )
1.  changes ← true
2.  while changes
3.      do changes ← false
4.          for def1 ∈ MethodDefSet
5.              do for def2 ∈ MethodDefSet
6.                  do if def1 ≠ def2 and moreSpecific(def1,def2)
7.                      then MethodDefSet ← MethodDefSet - {def1}
8.                          changes ← true
9.  return MethodDefSet

```

Figure 10.33 Remove Less Specific Method Definitions.

After we have reduced the set of possible method definitions down to a single definition, semantic analysis is almost complete. We must check for the following special cases of method calls:

- If an unqualified method call appears in a static context (the body of a static method, a static initializer, or an initialization expression for a static variable), then the method called must be static. (This is because the object reference symbol `this` is undefined in static contexts).
- Method calls qualified by a class name (`className.method`) must be to a static method.

- A call to a method that return void may not appear in an expression context (where a value is expected).

The complete process of checking a method call, as developed above, is defined in Figure 10.34.

```

callNode.Semantics( )
1.  qualifier.ExprSemantics()
2.  methodSet ← getMethods()
3.  args.Semantics()
4.  actualArgsType ← getArgTypes(args)
5.  for def ∈ methodSet
6.      do if not applicable(def.argsType,actualArgsType)
7.          then methodSet ← methodSet – {def}
8.  throwsSet ← union(qualifier.throwsSet, args.throwsSet)
9.  terminatesNormally ← true
10. if size(methodSet) = 0
11.     then GenerateErrorMessage("No Method matches this Call")
12.         return
13. elseif size(methodSet) > 1
14.     then methodSet ← filterDefs(methodSet)
15. if size(methodSet) > 1
16.     then GenerateErrorMessage("More than One Method matches this Call")
17. elseif inStaticContext() and methodSet.member.accessMode ≠ static
18.     then GenerateErrorMessage("Method Called Must Be Static")
19. elseif inExpressionContext() and methodSet.member.returnType = Void
20.     then GenerateErrorMessage("Call Must Return a Value")
21.     else throwsSet ← union(throwsSet, methodSet.member.declaredThrowsList)

```

Figure 10.34 Semantic Checks for a Method Call

As an example of how method calls are checked, consider the call of $M(arg)$ in following code fragment

```

class A { void M(A parm) {...}
         void M() {...} }
class B extends A { void M(B parm) {...}
                   void test(B arg) {M(arg);}}

```

In method $test$, where $M(arg)$ appears, three definitions of M are visible. All are accessible. Two of the three (those that take one parameter) are applicable to the call. The definition of $M(B\ parm)$ in B is more specific than the definition of $M(A\ parm)$ in A , so it is selected as the target of the call.

In all rules used to select among overloaded definitions, it is important to observe that the result type of a method is **never** used to decide if a definition is applicable. **Java** does not allow two method definitions with the same name that have identical parameters, but different result types to co-exist. Neither does **C++**. For example, the following two definitions force a multiple definition error:

```
int add(int i, int j) {...}
double add(int i, int j) {...}
```

This form of overloading is disallowed because it significantly complicates the process of deciding which overloaded definition to choose. Not only must the number and types of arguments be considered, but also the context within which result types are used. For example in

```
int i = 1 - add(2,3);
```

a semantic analyzer would have to conclude that the definition of `add` that returns a `double` is inappropriate because a `double`, subtracted from 1 would yield a `double`, which cannot be used to initialize an integer variable.

A few languages, like `Ada`, do allow overloaded method definitions that differ only in their result type. An analysis algorithm that can analyze this more general form of overloading may be found in [Baker 1982].

Interface and Constructor Calls. In addition to methods, `Java` allows calls to interfaces and constructors. The techniques we have developed apply directly to these constructs. An interface is an abstraction of a class, specifying a set of method definition without their implementations. For purposes of semantic analysis, implementations are unimportant. When a call to an interface is made, the methods declared in the interface (and perhaps its superinterfaces) are searched to find all declarations that are applicable. Once the correct declaration is identified, we can be sure that a corresponding implementation will be available at run-time.

Constructors are similar to methods in definition and structure. Constructors are called in object creation expressions (use of `new`) and in other constructors; they can never be called in expressions or statements. A constructor can be recognized by the fact that it has no result type (not even `void`). Once a constructor call is recognized as valid (by examining where it appears), the techniques developed above to select the appropriate declaration for a given call can be used.

Subprogram Calls in Other Languages. The chief difference between calls in `Java` and in languages like `C` and `C++` is that subprograms need not appear within classes. Rather, subprograms are defined at the global level (within a compilation unit). Languages like `Algol`, `Pascal` and `Modula 2` and `3` also allow subprograms to be declared locally, just like local variables and constants. Some languages allow overloading; other require a unique declaration for a given name.

Processing calls in these languages follows the same pattern as in `Java`. Using scoping and visibility rules, possible declarations corresponding to a given call are gathered. If overloading is disallowed, the nearest declaration is used. Other wise, a set of possible declarations is gathered. The number and type of arguments in the call is matched against the possible declarations. If a single suitable declaration isn't selected, a semantic error results.

11

Run-Time Storage Organization

The evolution of programming language design has led to the creation of increasingly sophisticated methods of run-time storage organization. Originally, all data was global, with a lifetime than spanned the entire program. Correspondingly, all storage allocation was *static*. A data object or instruction sequence was placed at a fixed address for the entire execution of a program.

Algol 60 and succeeding languages introduced local variables accessible only during the execution of a subprogram. This feature led to *stack allocation*. When a procedure was called, space for its local variables (its frame) was pushed on a run-time stack. Upon return, the space was popped. Only procedures actually executing were allocated space, and recursive procedures, which require multiple frames, were handled cleanly and naturally.

Lisp and later languages, including C, C++ and Java, popularized dynamically allocated data that could be created or freed at any time during execution.

Dynamic data led to *heap allocation*, which allows space to be allocated and freed at any time and in any order during program execution. With dynamic allocation, the number and size of data objects need not be fixed in advance. Rather, each program execution can “customize” its memory allocation.

All memory allocation techniques utilize the notion of a *data area*. A data area is a block of storage known by the compiler to have uniform storage allocation requirements. That is, all objects in a data area share the same data allocation policy. The global variables of a program can comprise a data area. Space for all variables is allocated when execution of a program begins, and variables remain allocated until execution terminates. Similarly, a block of data allocated by a call to `new` or `malloc` forms a single data area.

We’ll begin our study of memory allocation with static allocation in Section 11.1. In Section 11.2 we’ll investigate stack-based memory allocation. Finally, in Section 11.3, we’ll consider heap storage.

11.1 Static Allocation

In the earliest programming languages, including all assembly languages, as well as Cobol and Fortran, all storage allocation was static. Space for data objects was allocated in a fixed location for the lifetime of a program. Use of static allocation is feasible only when the number and size of all objects to be allocated is known at compile-time. This allocation approach, of course, makes storage allocation almost trivial, but it can also be quite wasteful of space. As a result, programmers must sometimes *overlay* variables. Thus, in Fortran, the `equivalence` statement

is commonly used to reduce storage needs by forcing two variables to share the same memory locations. (The C/C++ `union` can do this too.) Overlaying can lead to subtle programming errors, because assignment to one variable implicitly changes the value of another. Overlaying also reduces program readability.

In more modern languages, static allocation is used both for global variables that are fixed in size and accessible throughout program execution and for program literals (that is, constants) that need to be fixed throughout execution. Static allocation is used for `static` and `extern` variables in C/C++ and for `static` fields in Java classes. Static allocation is also routinely used for program code, since fixed run-time addresses are used in branch and call instructions. Also, since flow of control within a program is very hard to predict, it difficult to know which instructions will be needed next. Accordingly, if code is statically allocated, any execution order can be accommodated. Java allows classes to be dynamically loaded or compiled; but once program code is made executable, it is static.

Conceptually, we can bind static objects to absolute addresses. Thus if we generate an assembly language translation of a program, a global variable or program statement can be given a symbolic label that denotes a fixed memory address. It is often preferable to address a static data object as a pair (`DataArea`, `Offset`). `Offset` is fixed at compile-time, but the address of `DataArea` can be deferred to link- or run-time. In Fortran, for example, `DataArea` can be the start of one of many common blocks. In C, `DataArea` can be the start of a block of storage for the variables local to a module (a “.c” file). In Java, `DataArea` can be the start of a block of storage for the static fields of a class. Typically these addresses are bound when the

program is linked. Address binding must be deferred until link-time or run-time because subroutines and classes may be compiled independently, making it impossible for the compiler to know about all the data areas in a program.

Alternatively, the address of `DataArea` can be loaded into a register (the **global pointer**), which allows a static data item to be addressed as (**Register, Offset**). This addressing form is available on almost every machine. The advantage of addressing a piece of static data with a global pointer is that we can load or store a global value in one instruction. Since global addresses occupy 32 (or 64) bits, they normally “don’t fit” in a single instruction. Instead, lacking a global pointer, global addresses must be formed in several steps, first loading the high-order bits, then masking in the remain low-order bits.

11.2 Stack Allocation

Almost all modern programming languages allow recursive procedures, functions or methods, which require dynamic allocation. Each recursive call requires the allocation of a new copy of a routine’s local variables; thus the number of data objects required during program execution is not known at compile-time. To implement recursion, all the data space required for a routine (a procedure, function or method) is treated as a data area that, because of the special way it is handled, is called a *frame* or *activation record*.

Local data, held within a frame, is accessible only while a subprogram is active. In mainstream languages like C, C++ and Java, subprograms must return in a stack-like manner; the most recently called subprogram will be the first to

return. This means a frame may be pushed onto a *run-time stack* when a routine is called (activated). When the routine returns, the frame is popped from the stack, freeing the routine's local data. To see how stack allocation works, consider the C subprogram shown in Figure 11.1.

```
p(int a) {  
    double b;  
    double c[10];  
    b = c[a] * 2.51;  
}
```

Figure 11.1 A Simple Subprogram

Procedure `p` requires space for the parameter `a` as well as the local variables `b` and `c`. It also needs space for control information, such as the return address (a procedure may be called from many different places). As the procedure is compiled, the space requirements of the procedure are recorded (in the procedure's symbol table). In particular, the *offset* of each data item relative to the beginning of the frame is stored in the symbol table. The total amount of space needed, and thus the size of the frame, is also recorded. The memory requirements for individual variables (and hence an entire frame) is machine-dependent. Different architectures may assume different sizes for primitive values like integers or addresses.

In our example, assume `p`'s control information requires 8 bytes (this size is usually the same for all routines). Assume parameter `a` requires 4 bytes, local variable `b` requires 8 bytes, and local array `c` requires 80 bytes. Because many machines require that word and doubleword data be *aligned*, it is common practice to pad a frame (if necessary) so that its size is a multiple of 4 or 8 bytes. This

guarantees a useful invariant—at all times the top of the stack is properly aligned.

Figure 11.2 shows p 's frame.

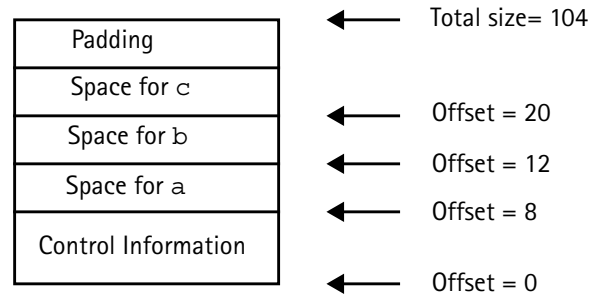


Figure 11.2 Frame for Procedure P

Within p , each local data object is addressed by its offset relative to the start of the frame. This offset is a fixed constant, determined at compile-time. Because we normally store the start of the frame in a register, each piece of data can be addressed as a (Register, Offset) pair, which is a standard addressing mode in almost all computer architectures. For example, if register R points to the beginning of p 's frame, variable b can be addressed as $(R, 12)$, with 12 actually being added to the contents of R at run-time, as memory addresses are evaluated.

Normally, the literal 2.51 of procedure p is not stored in p 's frame because the values of local data that are stored in a frame disappear with it at the end of a call. If 2.51 were stored in the frame, its value would have to be initialized before each call. It is easier and more efficient to allocate literals in a static area, often called a **literal pool** or **constant pool**. Java uses a constant pool to store literals, type, method and interface information as well as class and field names.

11.2.1 Accessing Frames at Run-Time

At any time during execution there can be many frames on the stack. This is because when a procedure A calls a procedure B, a frame for B's local variables is pushed on the stack, covering A's frame. A's frame can't be popped off because A will resume execution after B returns. In the case of recursive routines there can be hundreds or even thousands of frames on the stack. All frames but the topmost represent suspended subroutines, waiting for a call to return.

The topmost frame is *active*, and it is important to be able to access it directly. Since the frame is at the top of the stack, the stack top register could be used to access it. The run-time stack may also be used to hold data other than frames, like temporary or return values too large to fit within a register (arrays, structs, strings, etc.)

It is therefore unwise to require that the currently active frame always be at *exactly* the top of the stack. Instead a distinct register, often called the *frame pointer*, is used to access the current frame. This allows local variables to be accessed directly as `offset + frame pointer`, using the indexed addressing mode found on all modern machines.

As an example, consider the following recursive function that computes factorials.

```

int fact(int n) {
    if (n > 1)
        return n * fact(n-1);
    else return 1;
}

```

The run-time stack corresponding to the call `fact(3)` is shown in Figure 11.3 at the point where the call of `fact(1)` is about to return. In our example we show a slot for the function's return value at the very beginning of the frame. This means that upon return, the return value is conveniently placed on the stack, just beyond the end of the caller's frame. As an optimization, many compilers try to return scalar values in specially designated registers. This helps to eliminate unnecessary loads and stores. For function values too large to fit in a register (e.g., a `struct`), the stack is the natural choice.

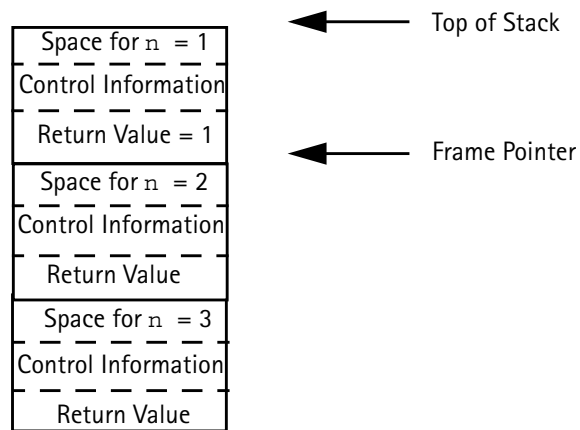


Figure 11.3 Run-time Stack for a Call of `fact(3)`

When a subroutine returns, its frame must be popped from the stack and the frame pointer must be reset to point to the caller's frame. In simple cases this can be done by adjusting the frame pointer by the size of the current frame. Because the stack may contain more than just frames (e.g., function return values or registers saved across calls), it is common practice to save the caller's frame pointer as part of the callee's control information. Thus each frame points to the preceding frame on the stack. This pointer is often called a *dynamic link* because it links a frame to its dynamic (run-time) predecessor. The run-time stack corresponding to a call of `fact(3)`, with dynamic links included, is shown in Figure 11.4.

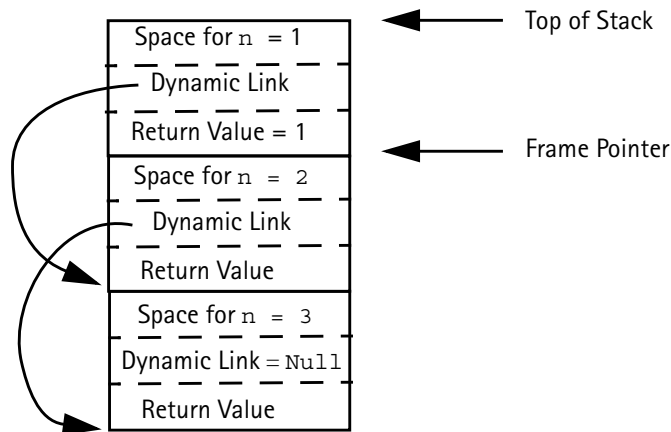


Figure 11.4 Run-time Stack for a Call of `fact(3)` with Dynamic Links

11.2.2 Handling Classes and Objects

C, C++ and Java do not allow procedures or methods to nest. That is, a procedure may not be declared within another procedure. This simplifies run-time data access—all variables are either global or local to the currently executing procedure.

Global variables are statically allocated. Local variables are part of a single frame, accessed through the frame pointer.

Languages often need to support simultaneous access to variables in multiple scopes. Java and C++, for example, allows classes to have member functions that have direct access to all instance variables. Consider the following Java class.

```
class k {  
    int a;  
    int sum() {  
        int b;  
        return a+b;  
    }  
}
```

Each object that is an instance of class `k` contains a member function `sum`. Only one translation of `sum` is created; it is shared by all instances of `k`. When `sum` executes it requires two pointers to access local and object-level data. Local data, as usual, resides in a frame on the run-time stack. Data values for a particular instance of `k` are accessed through an object pointer (called the `this` pointer in Java and C++). When `obj.sum()` is called, it is given an extra implicit parameter that a pointer to `obj`. This is illustrated in Figure 11.5. When `a+b` is computed, `b`, a local variable, is accessed directly through the frame pointer. `a`, a member of object `obj`, is accessed indirectly through the object pointer that is stored in the frame (as all parameters to a method are).

C++ and Java also allow **inheritance** via **subclassing**. That is, a new class can extend an existing class, adding new fields and adding or redefining methods. A

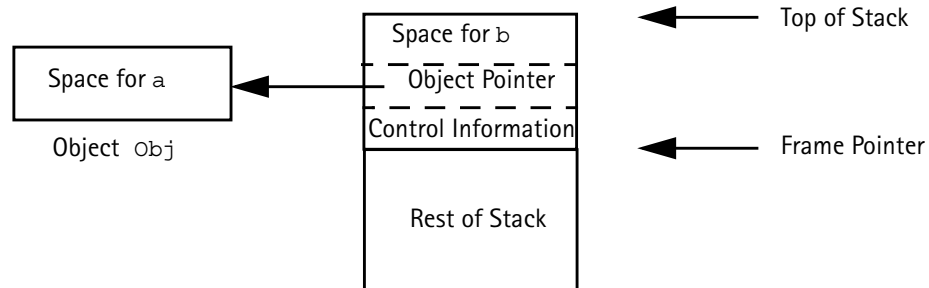


Figure 11.5 Accessing Local and Member Data in Java

subclass `D`, of class `C`, maybe be used in contexts expecting an object of class `C` (e.g., in method calls). This is supported rather easily—objects of class `D` always contain a class `C` object within them. That is, if `C` has a field `F` within it, so does `D`. The fields `D` declares are merely appended at the end of the allocations for `C`. As a result, access to fields of `C` within a class `D` object works perfectly. In Java, class `Object` is often used as a placeholder when an object of unknown type is expected. This works because all objects are subclasses of `Object`.

Of course, the converse cannot be allowed. A class `C` object may not be used where a class `D` object is expected, since `D`'s fields aren't present in `C`.

11.2.3 Handling Multiple Scopes

Many languages, including Ada, Pascal, Algol and Modula-3 allow procedure declarations to nest. The latest releases of Java allow classes to nest (see Exercise 6). Procedure nesting can be very useful, allowing a subroutine to directly access another routine's locals and parameters. However, run-time data structures are complicated because multiple frames, corresponding to nested procedure declara-

tions, may need to be accessed. To see the problem, assume that routines *can* nest in Java or C, and consider the following code fragment

```
int p(int a){
    int q(int b){
        if (b < 0)
            q(-b);
        else return a+b;
    }
    return q(-10);
}
```

When *q* executes, it can access not only its own frame, but also that of *p*, in which it is nested. If the depth of nesting is unlimited, so is the number of frames that must be made accessible. In practice, the level of procedure nesting actually seen is modest—usually no greater than two or three.

Two approaches are commonly used to support access to multiple frames. One approach generalizes the idea of dynamic links introduced earlier. Along with a dynamic link, we'll also include a *static link* in the frame's control information area. The static link will point to the frame of the procedure that statically encloses the current procedure. If a procedure is not nested within any other procedure, its static link is `null`. This approach is illustrated in Figure 11.6.

As usual, dynamic links always point to the next frame down in the stack. Static links always point down, but they may skip past many frames. They always point to the most recent frame of the routine that statically encloses the current

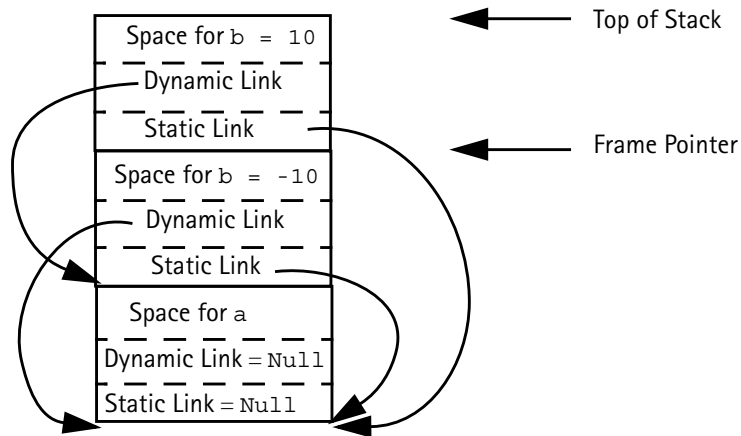


Figure 11.6 An Example of Static Links

routine. Thus in our example, the static links of both of q 's frames point to p , since it is p that encloses q 's definition. In evaluating the expression $a+b$ that q returns, b , being local to q , is accessed directly through the frame pointer. Variable a is local to p , but also visible to q because q nests within p . a is accessed by extracting q 's static link, then using that address (plus the appropriate offset) to access a . (See Exercise 18.)

An alternative to using static links to access frames of enclosing routines is the use of a *display*. A display generalizes our use of a frame pointer. Rather than maintaining a single register, we maintain a *set of registers* which comprise the display. If procedure definitions nest n deep (this can be easily determined by examining a program's AST), we will need $n+1$ display registers. Each procedure definition is tagged with a nesting level. Procedures not nested within any other routine are at level 0. Procedures nested within only one routine are at level 1, etc. Frames for routines at level 0 are always accessed using display register D0. Those

at level 1 are always accessed using register D1, etc. Thus whenever a procedure r is executing, we have direct access to r 's frame plus the frames of all routines that enclose r . Each of these routines must be at a different nesting level, and hence will use a different display register. Consider Figure 11.7, which illustrates our earlier example, now using display registers rather than static links.

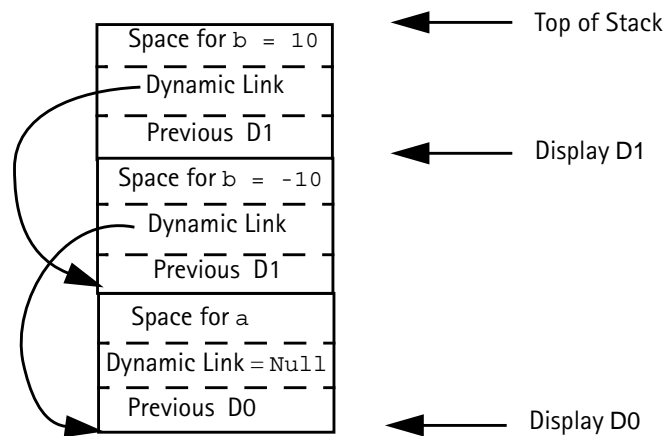


Figure 11.7 An Example of Display Registers

Since q is at nesting level 1, its frame is pointed to by D1. All of q 's local variables, including b , are at a fixed offset relative to D1. Similarly, since p is at nesting level 0, its frame and local variables are accessed via D0. Note that each frame's control information area contains a slot for the previous value of the frame's display register. This value is saved when a call begins and restored when the call ends. The dynamic link is still needed, because the previous display values doesn't always point to the caller's frame.

Not all compiler writers agree on whether static links or displays are better to use. Displays allow direct access to all frames, and thus make access to all visible

variables very efficient. However, if nesting is deep, several valuable registers may need to be reserved. Static links are very flexible, allowing unlimited nesting of procedures. However, access to non-local procedure variables can be slowed by the need to extract and follow static links.

Fortunately, the code generated using the two techniques can be improved. Static links are just address-valued expressions computed to access variables (much like address calculations involving pointer variables). A careful compiler can notice that an expression is being needlessly recomputed, and reuse a previous computation, often directly from a register. Similarly, a display can be allocated statically in memory. If a particular display value is used frequently, a register allocator will place the display value in a register to avoid repeated loads and stores (just as it would for any other heavily used variable).

11.2.4 Block-Level Allocation

Java, C and C++, as well as most other programming languages, allow declaration of local variables within blocks as well as within procedures. Often a block will contain only one or two variables, along with statements that use them. Do we allocate an entire frame for each such block?

We could, by considering a block with local variables to be an in-line procedure without parameters, to be allocated its own frame. This would necessitate a display or static links, even in Java or C, because blocks can nest. Further, execution of a block would become more costly, since frames need to be pushed and popped, display registers or static links updated, and so forth.

To avoid this overhead, it is possible to use frames only for true procedures, even if blocks within a procedure have local declarations. This technique is called *procedure-level frame allocation*, as contrasted with *block-level frame allocation*, which allocates a frame for each block that has local declarations.

The central idea of procedure-level frame allocation is that the relative location of variables in individual blocks within a procedure can be computed and fixed at compile-time. This works because blocks are entered and exited in a strictly textual order. Consider, the following procedure

```
void p(int a) {  
    int b;  
    if (a > 0)  
        {float c, d;  
         // Body of block 1 }  
    else {int e[10];  
         // Body of block 2 }  
}
```

Parameter `a` and local variable `b` are visible throughout the procedure. However the `then` and `else` parts of the `if` statement are mutually exclusive. Thus variables in block 1 and block 2 can *overlay* each other. That is, `c` and `d` are allocated just beyond `b` as is the array `e`. Because variables in both blocks can't ever be accessed at the same time, this overlaying is safe. The layout of the frame is illustrated in Figure 11.8.

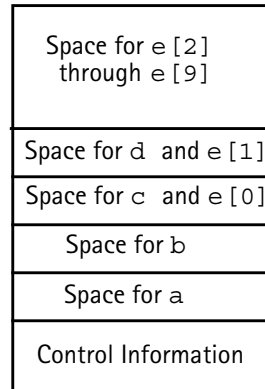


Figure 11.8 An Example of a Procedure-Level Frame

Offsets for variables within a block are assigned just after the last variable in the enclosing scope within the procedure. Thus both `c` and `e []` are placed after `b` because both block 1 and block 2 are enclosed by the block comprising `p`'s body. As blocks are compiled a “high water mark” is maintained that represents the maximum offset used by any local variable. This high water mark determines the size of the overall frame. Thus `a [9]` occupies the maximum offset within the frame, so its location determines the size of `p`'s frame.

The process of assigning local variables procedure-level offsets is sometimes done using **scope flattening**. That is, local declarations are mapped to equivalent procedure-level declarations. This process is particularly effective if procedure-level register allocation is part of the compilation process (see Section 15.3).

11.2.5 More About Frames

ClosuresIn C it is possible to create a pointer to a function. Since a function's frame is created only when it is called, a function pointer is implemented as the

function's entry point address. In C++, pointers to member functions of a class are allowed. When the pointer is used, a particular instance of the class must be provided by the user program. That is, two pointers are needed, one to the function itself and a second pointer to the class instance in which it resides. This second pointer allows the member function to access correctly local data belonging to the class.

Other languages, particularly functional languages like Lisp, Scheme, and ML, are much more general in their treatment of functions. Functions are *first-class objects*. They can be stored in variables and data structures, created during execution, and returned as function results.

Run-time creation and manipulation of functions can be extremely useful. For example, it is sometimes the case that computation of $f(x)$ takes a significant amount of time. Once $f(x)$ is known, it is a common optimization, called **memoizing**, to table the pair $(x, f(x))$ so that subsequent calls to f with argument x can use the known value of $f(x)$ rather than recompute it. In ML it is possible to write a function `memo` that takes a function `f` and an argument `arg`. `memo` computes $f(arg)$ *and* also returns a “smarter” version of `f` that has the value of $f(arg)$ “built into” it. This smarter version of `f` can be used instead of `f` in all subsequent computations.

```
fun memo(fct,parm)= let val ans = fct(parm) in
  (ans, fn x=> if x=parm then ans else fct(x))end;
```

When the version of `fact` returned by `memo` is called, it will access to the values of `parm`, `fact` and `ans`, which are used in its definition. After `memo` returns, its frame must be preserved since that frame contains `parm`, `fact` and `ans` within it.

In general when a function is created or manipulated, we must maintain a pair of pointers. One is to the machine instructions that implement the function, and the other is to the frame (or frames) that represent the function's execution environment. This pair of pointers is called a **closure**. Note also that when functions are first-class objects, a frame corresponding to a call may be accessed *after* the call terminates. This means frames can't always be allocated on the run-time stack. Instead, they are allocated in the heap and garbage-collected, just like user-created data. This appears to be inefficient, but Appel [1987] has shown that in some circumstances heap allocation of frames can be faster than stack allocation.

Cactus Stacks Many programming languages allow the concurrent execution of more than one computation in the same program. Units of concurrent execution are sometimes called tasks, or processes or threads. In some cases a new system-level process is created (as in the case of `fork` in C). Because a great deal of operating system overhead is involved, such processes are called *heavy-weight processes*. A less expensive alternative is to execute several threads of control in a single system-level process. Because much less "state" is involved, computations that execute concurrently in a single system process are called *light-weight processes*.

A good example of light-weight processes are *threads* in Java. As illustrated below, a Java program may initiate several calls to member functions that will execute simultaneously.

```
public static void main (String args[]) {  
    new AudioThread("Audio").start();  
    new VideoThread("Video").start();  
}
```

Here two instances of Thread subclasses are started, and each executes concurrently with the other. One thread might implement the audio portion of an application, while the other implements the video portion.

Since each thread can initiate its own sequence of calls (and possibly start more threads), all the resulting frames can't be pushed on a single run-time stack (the exact order in which threads execute is unpredictable). Instead, each thread gets its own stack segment in which frames it creates may be pushed. This stack structure is sometimes called a *cactus stack*, since it is reminiscent of the saguaro cactus, which sends out arms from the main trunk and from other arms. It is important that the thread handler be designed so that segments are properly deallocated when their thread is terminated. Since Java guarantees that all temporaries and locals are contained within a method's frame, stack management is limited to proper allocation and deallocation of frames.

A Detailed Frame Layout The layout of a frame is usually standardized for a particular architecture. This is necessary to support calls to subroutines translated by dif-

ferent compilers. Since languages and compilers vary in the features they support, the frame layout chosen as the standard must be very general and inclusive. As an example, consider Figure 11.9, which illustrates the frame layout used by the MIPS architecture.

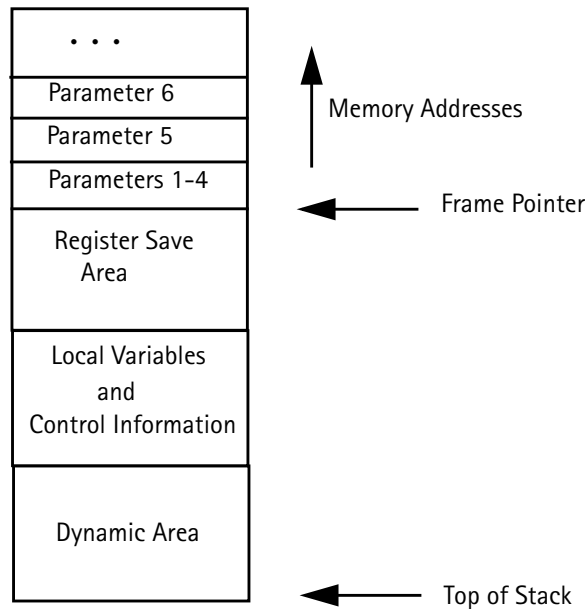


Figure 11.9 Layout for MIPS R3000

By convention, the first four parameters, if they are scalar, are passed in registers. Additional parameters, as well as non-scalar by-value parameters, are passed through the stack. The slots for parameters 1-4 can be used to save parameter registers when a call is made from within a procedure. The register save area is used at two different times. Registers are commonly partitioned in **caller-saved** registers (which a caller is responsible for) and **callee-saved** registers (which a subprogram is responsible for). When execution of a subroutine begins, callee-saved registers

used by the subroutine are saved in the register save area. When a call is made from within the subroutine, caller-saved registers that are in use are saved in the register save area. At different call sites different registers may be in use. The register save area must be big enough to handle all calls within a particular subroutine. Often a fixed size register save area, big enough to accommodate all caller-saved and callee-saved registers, is used. This may waste a bit of space, but only registers actually in use are saved.

The local variables and control information area contains space for all local variables. It also contains space for the return address register, and the value of the caller's frame pointer. The value of a static link or display register may be saved here if they are needed. The stack top may be reset, upon return, by adding the size of the parameter area to the frame pointer. (Note that on the MIPS, as well as on many other computers, the stack grows downward, from high to low addresses).

The details of subroutine calls are explored more thoroughly in Section 13.2 (at the bytecode level) and Section 15.1 (at the machine code level).

Because the Java Virtual Machine is designed to run on a wide variety of architectures, the exact details of its run-time frame layout are unspecified. A particular implementation (such as the JVM running on a MIPS processor), chooses a particular layout, similar to that shown in Figure 11.9.

Some languages allow the size of a frame to be expanded during execution. In C, for example, `alloca` allocates space on demand on the stack. Space is pushed beyond the end of the frame. Upon return, this space is automatically freed when the frame is popped.

Some languages allow the creation of *dynamic arrays* whose bounds are set at run-time when a frame is pushed (e.g., `int data [max (a, b)]`). At the start of a subprogram's execution, array bounds are evaluated and necessary space is pushed in the dynamic area of the frame.

C and C++ allow subroutines like `printf` and `scanf` to have a variable number of arguments. The MIPS frame design supports such routines, since parameter values are placed, in order, just above the frame pointer.

Non-scalar return values can be handled by treating the return value as the “0-th parameter.” As an optimization, calls to functions that return a non-scalar result sometimes pass an address as the 0-th parameter. This represents a place where the return value can be stored prior to return. Otherwise, the return value is left on the stack by the function.

11.3 Heap Management

The most flexible storage allocation mechanism is *heap allocation*. Any number of data objects can be allocated and freed at any time and in any order. A storage pool, usually called a *heap*, is used. Heap allocation is enormously popular—it is difficult to imagine a non-trivial Java or C program that does not use `new` or `malloc`.

Heap allocation and deallocation is far more complicated than is the case for static or stack allocation. Complex mechanisms may be needed to satisfy a request for space. Indeed, in some cases, all of the heap (many megabytes) may need to be examined. It takes great care to make heap management fast and efficient.

11.3.1 Allocation Mechanisms

A request for heap space may be *explicit* or *implicit*. An explicit request involves a call to a routine like `new` or `malloc`, with a request for a specific number of bytes. An explicit pointer or reference to the newly allocated space is returned (or a `null` pointer if the request could not be honored).

Some languages allow the creation of data objects of unknown size. Assume that in C++, as in Java, the `+` operator is overloaded to represent string catenation. That is, the expression `Str1 + Str2` creates a new string representing the catenation of strings `Str1` and `Str2`. There is no compile-time bound on the sizes of `Str1` and `Str2`, so heap space must be allocated to hold the newly created string.

Whether allocation is explicit or implicit, a *heap allocator* is needed. This routine takes a size parameter and examines unused heap space to find free space that satisfies the request. A *heap block* is returned. This block will be big enough to satisfy the space request, but it may well be bigger. Allocated heap blocks are almost always single- or double-word aligned to avoid alignment problems in heap-allocated arrays or class instances. Heap blocks contain a header field (usually a word) that contains the size of the block as well as auxiliary bookkeeping information. (The size information is necessary to properly “recycle” the block if it is later deallocated.) A minimum heap block size (commonly 16 bytes) is usually imposed to simplify bookkeeping and guarantee alignment.

The complexity of heap allocation depends in large measure on how *deallocation* is done. Initially, the heap is one large block of unallocated memory. Memory

requests can be satisfied by simply modifying an “end of heap” pointer, very much as a stack is pushed by modifying a stack pointer. Things get more involved when previously allocated heap objects are deallocated and reused. Some deallocation techniques *compact* the heap, moving all “in use” objects to one end of the heap. This means unused heap space is always contiguous, making allocation (via a heap pointer) almost trivial.

Some heap deallocation algorithms have the useful property that their speed depends not on the total number of heap objects allocated, but rather only on those objects still in use. If most heap objects “die” soon after their allocation (and this does seem to often be the case), deallocation of these objects is essentially free.

Unfortunately, many deallocation techniques *do not* perform compaction. Deallocated objects must be stored for future reuse. The most common approach is to create a *free space list*. A free space list is a linked (or doubly-linked) list that contains all the heap blocks known not to be in use. Initially it contains one immense block representing the entire heap. As heap blocks are allocated, this block shrinks. When heap blocks are returned, they are appended to the free space list.

The most common way of maintaining the free space list is to append blocks to the head of the list as they are deallocated. This simplifies deallocation a bit, but makes *coalescing* of free space difficult.

It often happens that two blocks, physically adjacent in the heap, are eventually deallocated. If we can recognize that the two blocks are now both free and

adjacent, they can be coalesced into one larger free block. One large block is preferable to two smaller blocks, since the combined block can satisfy requests too large for either of the individual blocks.

The *boundary tags* approach (Knuth 1973) allows us to identify and coalesce adjacent free heap blocks. Each heap block, whether allocated or on the free space list, contains a tag word on both of its ends. This tag word contains a flag indicating “free” or “in use” and the size of the block. When a block is freed, the boundary tags of its neighbors are checked. If either or both neighbors are marked as free, they are unlinked from the free space list and coalesced with the current free block.

A free space list may also be kept in *address order*; that is, sorted in order of increasing heap addresses. Boundary tags are no longer needed to identify adjacent free blocks, though maintenance of the list in sorted order is now more expensive.

When a request for n bytes of heap space is received, the heap allocator must search the free space list for a block of sufficient size. But how much of the free space list is to be searched? (It may contain many thousands of blocks.) What if no free block exactly matches the current request? There are many approaches that might be used. We’ll consider briefly a few of the most widely used techniques.

Best FitThe free space list is searched, perhaps exhaustively, for the free block that matches most closely the requested size. This minimizes wasted heap space, though it may create tiny fragments too small to be used very often. If the free

space list is very long, a best fit search may be quite slow. Segregated free space lists (see below) may be preferable.

First FitThe first free heap block of sufficient size is used. Unused space within the block is split off and linked as a smaller free space block. This approach is fast, but may “clutter” the beginning of the free space list with a number of blocks too small to satisfy most requests.

Next FitThis is a variant of first fit in which succeeding searches of the free space list begin at the position where the last search ended, rather than at the head of the list. The idea is to “cycle through” the entire free space list rather than always revisiting free blocks at the head of the list. This approach reduces *fragmentation* (in which blocks are split into small, difficult to use pieces). However, it also reduces *locality* (how densely packed active heap objects are). If we allocate heap objects that are widely distributed throughout the heap, we may increase cache misses and page faults, significantly impacting performance.

Segregated Free Space ListsThere is no reason why we must have only *one* free space list. An alternative is to have several, indexed by the size of the free blocks they contain. Experiments have shown that programs frequently request only a few “magic sizes.” If we divide the heap into segments, each holding only one size, we can maintain individual free space lists for each segment. Because heap object size is fixed, no headers are needed.

A variant of this approach is to maintain lists of objects of special “strategic” sizes (16, 32, 64, 128, etc.) When a request for size s is received, a block of the smallest size $\leq s$ is selected (with excess size unused within the allocated block).

Another variant is to maintain a number of free space lists, each containing a range of sizes. When a request for size s is received, the free space list covering s 's range is searched using a best fit strategy.

Fixed-Size Subheaps Rather than linking free objects onto lists according to their size, we can divide the heap into a number of *subheaps*, each allocating objects of a single fixed size. We can then use a *bitmap* to track an object's allocation status. That is, each object is mapped to a single bit in a large array. A 1 indicates the object is in use; a 0 indicates it is free. We need no explicit headers or free-space lists. Moreover, since all objects are of the same size, *any* object whose status bit is 0 may be allocated. We do have the problem though that subheaps may be used unevenly, with one nearly exhausted while another is lightly-used.

11.3.2 Deallocation Mechanisms

Allocating heap space is fairly straightforward. Requests for space are satisfied by adjusting an end-of-heap pointer, or by searching a free space list. But how do we deallocate heap memory no longer in use? Sometimes we may never need to deallocate! If heap objects are allocated infrequently or are very long-lived, deallocation is unnecessary. We simply fill heap space with “in use” objects.

Unfortunately, many—perhaps most—programs cannot simply ignore deallocation. Experience shows that many programs allocate huge numbers of short-

lived heap objects. If we “pollute” the heap with large numbers of **dead** objects (that are no longer accessible), locality can be severely impacted, with active objects spread throughout a large address range. Long-lived or continuously running programs can also be plagued by **memory leaks**, in which dead heap objects slowly accumulate until a program’s memory needs exceed system limits.

User-controlled Deallocation Deallocation can be manual or automatic. Manual deallocation involves explicit programmer-initiated calls to routines like `free(p)` or `delete(p)`. Pointer `p` identifies the heap object to be freed. The object’s size is stored in its header. The object may be merged with adjacent unused heap objects (if boundary tags or an address-ordered free space list is used). It is then added to a free-space list for subsequent reallocation.

It is the programmer’s responsibility to free unneeded heap space by executing deallocation commands. The heap manager merely keeps track of freed space and makes it available for later reuse. The really hard decision—when space should be freed—is shifted to the programmer, possibly leading to catastrophic *dangling pointer* errors. Consider the following C program fragment

```
q = p = malloc(1000);  
  
free(p);  
  
/* code containing a number of malloc's */  
  
q[100] = 1234;
```

After `p` is freed, `q` is a dangling pointer. That is, `q` points to heap space that is no longer considered allocated. Calls to `malloc` may reassign the space pointed to

by q . Assignment through q is illegal, but this error is almost never detected. Such an assignment may change data that is now part of another heap object, leading to very subtle errors. It may even change a header field or a free-space link, causing the heap allocator itself to fail.

11.3.3 Automatic Garbage Collection

The alternative to manual deallocation of heap space is automatic deallocation, commonly called *garbage collection*. Compiler-generated code and support subroutines track pointer usage. When a heap object is no longer pointed to (that is, when it is *garbage*), the object is automatically deallocated (collected) and made available for subsequent reuse.

Garbage collection techniques vary greatly in their speed, effectiveness and complexity. We shall consider briefly some of the most important approaches. For a more thorough discussion see [Wilson 96] or [JL 96].

Reference Counting One of the oldest and simplest garbage collection techniques is *reference counting*. A field is added to the header of each heap object. This field, the object's *reference count*, records how many references (pointers) to the heap object exist. When an object's reference count reaches zero, it is garbage and may be added to the free-space list for future reuse.

The reference count field must be updated whenever a reference is created, copied, or destroyed. When a procedure returns, the reference counts for all objects pointed to by local variables must be decremented. Similarly, when a refer-

ence count reaches zero and an object is collected, all pointers in the collected object must also be followed and corresponding reference counts decremented.

As shown in Figure 11.10, reference counting has particular difficulty with *circular structures*. If pointer P is set to null, the object's reference count is reduced to 1. Now both objects have a non-zero count, but neither is accessible through any external pointer. That is, the two objects are garbage, but won't be recognized as such.

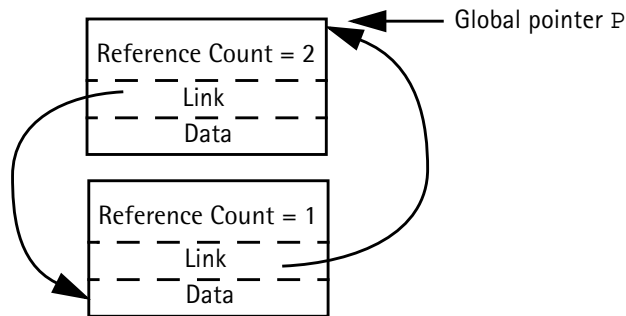


Figure 11.10 An Example of Cyclic Structures

If circular structures are rare, this deficiency won't be much of a problem. If they are common, then an auxiliary technique, like mark-sweep collection, will be needed to collect garbage that reference counting misses.

An important aspect of reference counting is that it is *incremental*. That is, whenever a pointer is manipulated, a small amount of work is done to support garbage collection. This is both an advantage and a disadvantage. It is an advantage in that the cost of garbage collection is smoothly distributed throughout a computation. A program doesn't need to stop when heap space grows short and do a lengthy collection. This can be crucial when fast real-time response is

required. (We don't want the controls of an aircraft to suddenly "freeze" for a second or two while a program collects garbage!)

The incremental nature of reference counting can also be a disadvantage when garbage collection isn't really needed. If we have a complex data structure in which pointers are frequently updated, but in which few objects ever are discarded, reference counting always adjusts counts that rarely, if ever, go to zero.

How big should a reference count field be? Interestingly, experience has shown that it doesn't need to be particularly large. Often only a few bits suffice. The idea here is that if a count ever reaches the maximum representable count (perhaps 7 or 15 or 31), we "lock" the count at that value. Objects with a locked reference count won't ever be detected as garbage by their counts, but they can be collected using other techniques when circular structures are collected.

In summary, reference counting is a simple technique whose incremental nature is sometimes useful. Because of its inability to handle circular structures and its significant per-pointer operation cost, other garbage collection techniques, as described below, are often more attractive alternatives.

Mark-Sweep Collection Rather than incrementally collecting garbage as pointers are manipulated, we can take a batch approach. We do nothing until heap space is nearly exhausted. Then we execute a *marking phase* which aims to identify all live (non-garbage) heap objects.

Starting with global pointers and pointers in stack frames, we mark reachable heap objects (perhaps setting a bit in the object's header). Pointers in marked heap objects are also followed, until all live heap objects are marked.

After the marking phase, we know that any object not marked is garbage that may be freed. We then *sweep* through the heap, collecting all unmarked objects and returning them to the free space list for later reuse. During the sweep phase we also clear all marks from heap objects found to be still in use.

Mark-sweep garbage collection is illustrated in Figure 11.11. Objects 1 and 3 are marked because they are pointed to by global pointers. Object 5 is marked because it is pointed to by object 3, which is marked. Shaded objects are not marked and will be added to the free-space list.

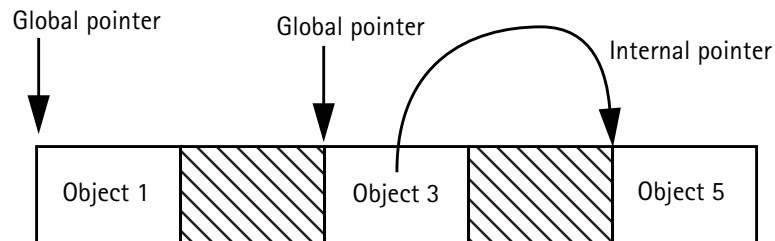


Figure 11.11 Mark-Sweep Garbage Collection

In any mark-sweep collector, it is vital that we mark *all* accessible heap objects. If we miss a pointer, we may fail to mark a live heap object and later incorrectly free it. Finding all pointers is not too difficult in languages like Lisp and Scheme that have very uniform data structures, but it is a bit tricky in languages like Java, C and C++, that have pointers mixed with other types within data structures, implicit pointers to temporaries, and so forth. Considerable infor-

mation about data structures and frames must be available at run-time for this purpose. In cases where we can't be sure if a value is a pointer or not, we may need to do conservative garbage collection (see below).

Mark-sweep garbage collection also has the problem that *all* heap objects must be swept. This can be costly if most objects are dead. Other collection schemes, like copying collectors, examine *only* live objects.

After the sweep phase, live heap objects are distributed throughout the heap space. This can lead to poor locality. If live objects span many memory pages, paging overhead may be increased. Cache locality may be degraded too.

We can add a *compaction phase* to mark-sweep garbage collection. After live objects are identified, they are placed together at one end of the heap. This involves another tracing phase in which global, local and internal heap pointers are found and adjusted to reflect the object's new location. Pointers are adjusted by the total size of all garbage objects between the start of the heap and the current object. This is illustrated in Figure 11.12.

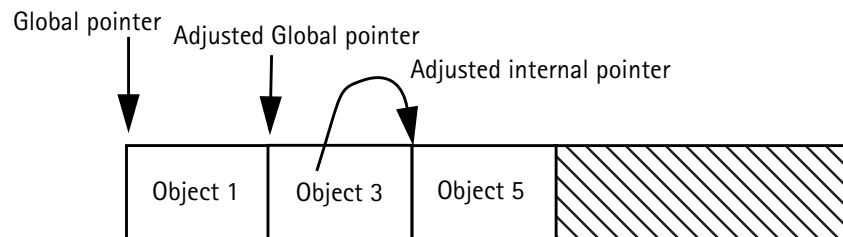


Figure 11.12 Mark-Sweep Garbage Collection with Compaction

Compaction is attractive because all garbage objects are merged together into one large block of free heap space. Fragments are no longer a problem. Moreover,

heap allocation is greatly simplified. An “end of heap” pointer is maintained. Whenever a heap request is received, the end of heap pointer is adjusted, making heap allocation no more complex than stack allocation.

However, because pointers must be adjusted, compaction may not be suitable for languages like C and C++, in which it is difficult to unambiguously identify pointers.

Copying Collectors Compaction provides many valuable benefits. Heap allocation is simple and efficient. There is no fragmentation problem, and because live objects are adjacent, paging and cache behavior is improved. An entire family of garbage collection techniques, called *copying collectors* have been designed to integrate copying with recognition of live heap objects. These copying collectors are very popular and are widely used, especially with functional languages like ML.

We’ll describe a simple copying collector that uses *semispaces*. We start with the heap divided into two halves—the *from* and *to spaces*. Initially, we allocate heap requests from the from space, using a simple “end of heap” pointer. When the from space is exhausted, we stop and do garbage collection.

Actually, though we *don’t* collect garbage. What we do is collect live heap objects—garbage is never touched. As was the case for mark-sweep collectors, we trace through global and local pointers, finding live objects. As each object is found, it is moved from its current position in the from space to the next available position in the to space. The pointer is updated to reflect the object’s new location. A “forwarding pointer” is left in the object’s old location in case there are

multiple pointers to the same object (we want only one object with all original pointers properly updated to the new location).

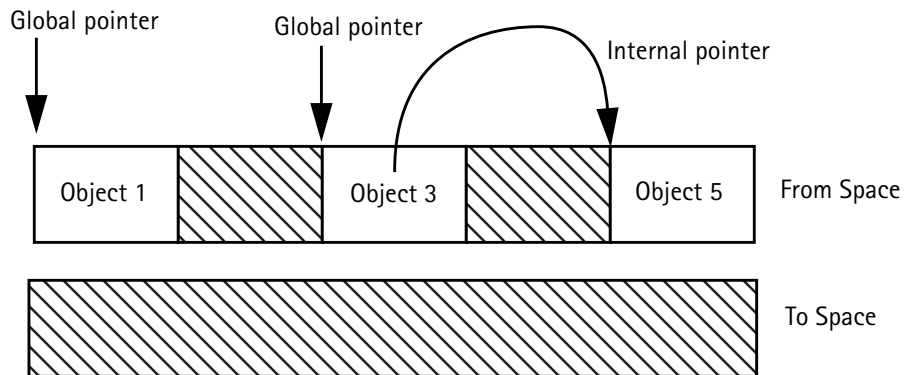


Figure 11.13 Copying Garbage Collection (a)

This is illustrated in Figure 11.13. The from space is completely filled. We trace global and local pointers, moving live objects to the to space and updating pointers. This is illustrated in Figure 11.14. (Dashed arrows are forwarding pointers). We have yet to handle pointers internal to copied heap objects. All copied heap objects are traversed. Objects referenced are copied and internal pointers are updated. Finally, the to and from spaces are interchanged, and heap allocation resumes just beyond the last copied object. This is illustrated in Figure 11.15.

The biggest advantage of copying collectors is their speed. Only live objects are copied; deallocation of dead objects is essentially free. In fact, garbage collection can be made, on average, as fast as you wish—simply make the heap bigger. As the heap gets bigger, the time between collections increases, reducing the number of times a live object must be copied. In the limit, objects are never copied, so garbage collection becomes free!

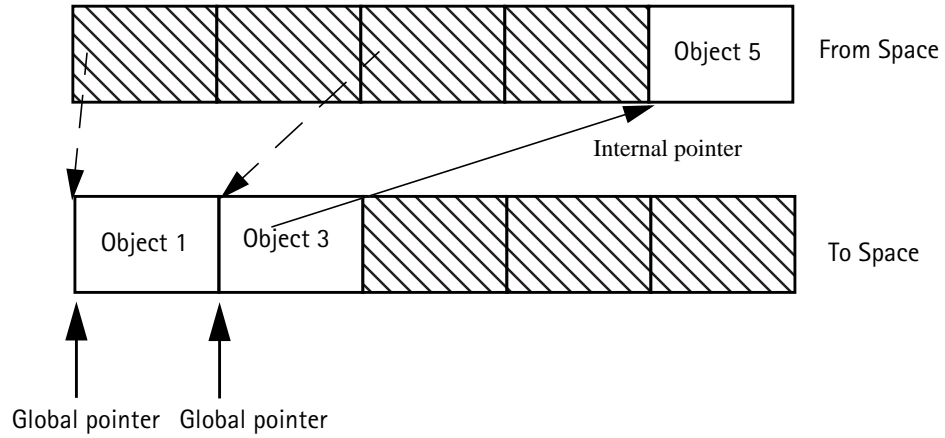


Figure 11.14 Copying Garbage Collection (b)

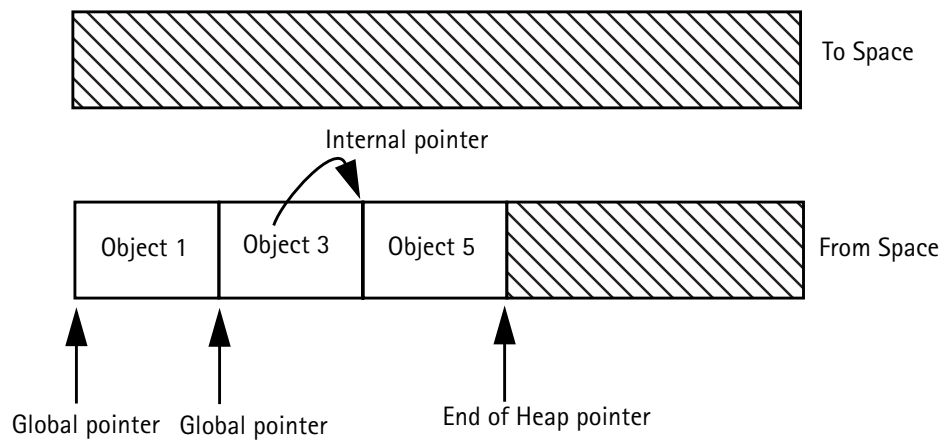


Figure 11.15 Copying Garbage Collection (c)

Of course, we can't increase the size of heap memory to infinity. In fact, we don't want to make the heap so large that paging is required, since swapping pages to disk is dreadfully slow. If we can make the heap large enough that the lifetime of most heap objects is less than the time between collections, then deallocation of short-lived objects will appear to be free, though longer-lived objects will still exact a cost.

Aren't copying collectors terribly wasteful of space? After all, at most only half of the heap space is actually used. The reason for this apparent inefficiency is that *any* garbage collector that does compaction must have an area to copy live objects to. Since in the worst case *all* heap objects could be live, the target area must be as large as the heap itself. To avoid copying objects more than once, copying collectors reserve a to space as big as the from space. This is essentially a space-time trade-off, making such collectors very fast at the expense of possibly wasted space.

If we have reason to believe that the time between garbage collections will be greater than the average lifetime of most heap objects, we can improve our use of heap space. Assume that 50% or more of the heap will be garbage when the collector is called. We can then divide the heap into 3 segments, which we'll call A, B and C. Initially, A and B will be used as the from space, utilizing 2/3 of the heap. When we copy live objects, we'll copy them into segment C, which will be big enough if half or more of the heap objects are garbage. Then we treat C and A as the from space, using B as the to space for the next collection. If we are unlucky and more than 1/2 the heap contains live objects, we can still get by. Excess objects are copied onto an auxiliary data space (perhaps the stack), then copied into A after all live objects in A have been moved. This slows collection down, but only rarely (if our estimate of 50% garbage per collection is sound). Of course, this idea generalizes to more than 3 segments. Thus if 2/3 of the heap were garbage (on average), we could use 3 of 4 segments as from space and the last segment as to space.

Generational TechniquesThe great strength of copying collectors is that they do not work for objects that are born and die between collections. However, not all heap objects are so short-lived. In fact, some heap objects are very long-lived. For example, many programs create a dynamic data structure at their start, and utilize that structure throughout the program. Copying collectors handle long-lived objects poorly. They are repeatedly traced and moved between semispaces without any real benefit.

Generational garbage collection techniques [Unger 1984] were developed to better handle objects with varying lifetimes. The heap is divided into two or more *generations*, each with its own to and from space. New objects are allocated in the youngest generation, which is collected most frequently. If an object survives across one or more collections of the youngest generation, it is “promoted” to the next older generation, which is collected less often. Objects that survive one or more collections of this generation are then moved to the next older generation. This continues until very long-lived objects reach the oldest generation, which is collected very infrequently (perhaps even never).

The advantage of this approach is that long-lived objects are “filtered out,” greatly reducing the cost of repeatedly processing them. Of course, some long-lived objects will die and these will be caught when their generation is eventually collected.

An unfortunate complication of generational techniques is that although we collect older generations infrequently, we must still trace their pointers in case they reference an object in a newer generation. If we don’t do this, we may mistake a

live object for a dead one. When an object is promoted to an older generation, we can check to see if it contains a pointer into a younger generation. If it does, we record its address so that we can trace and update its pointer. We must also detect when an existing pointer inside an object is changed. Sometimes we can do this by checking “dirty bits” on heap pages to see which have been updated. We then trace all objects on a page that is dirty. Otherwise, whenever we assign to a pointer that already has a value, we record the address of the pointer that is changed. This information then allows us to only trace those objects in older generations that might point to younger objects.

Experience shows that a carefully designed generational garbage collectors can be very effective. They focus on objects most likely to become garbage, and spend little overhead on long-lived objects. Generational garbage collectors are widely used in practice.

Conservative Garbage Collection The garbage collection techniques we’ve studied all require that we identify pointers to heap objects accurately. In strongly typed languages like Java or ML, this can be done. We can table the addresses of all global pointers. We can include a code value in a frame (or use the return address stored in a frame) to determine the routine a frame corresponds to. This allows us to then determine what offsets in the frame contain pointers. When heap objects are allocated, we can include a type code in the object’s header, again allowing us to identify pointers internal to the object.

Languages like C and C++ are weakly typed, and this makes identification of pointers much harder. Pointers may be type-cast into integers and then back into pointers. Pointer arithmetic allows pointers into the middle of an object. Pointers in frames and heap objects need not be initialized, and may contain random values. Pointers may overlay integers in unions, making the current type a dynamic property.

As a result of these complications, C and C++ have the reputation of being incompatible with garbage collection. Surprisingly, this belief is false. Using *conservative garbage collection*, C and C++ programs *can* be garbage collected.

The basic idea is simple—if we can't be sure whether a value is a pointer or not, we'll be conservative and assume it is a pointer. If what we think is a pointer isn't, we may retain an object that's really dead, but we'll find all valid pointers, and never incorrectly collect a live object. We may mistake an integer (or a floating value, or even a string) as a pointer, so compaction in any form *can't* be done. However, mark-sweep collection will work.

Garbage collectors that work with ordinary C programs have been developed [BW 1988]. User programs need not be modified. They simply are linked to different library routines, so that `malloc` and `free` properly support the garbage collector. When new heap space is required, dead heap objects may be automatically collected, rather than relying entirely on explicit `free` commands (though `free`s are allowed; they sometimes simplify or speed heap reuse).

With garbage collection available, C programmers need not worry about explicit heap management. This reduces programming effort and eliminates errors

in which objects are prematurely freed, or perhaps never freed. In fact, experiments have shown [Zorn 93] that conservative garbage collection is very competitive in performance with application-specific manual heap management.

Exercises

1. Show the frame layout corresponding to the following C function:

```
int f(int a, char *b){
    char c;
    double d[10];
    float e;
    ...
}
```

Assume control information requires 3 words and that `f`'s return value is left on the stack. Be sure to show the offset of each local variable in the frame and be sure to provide for proper alignment (ints and floats on word boundaries and doubles on doubleword boundaries).

2. Show the sequence of frames, with dynamic links, on the stack when `r(3)` is executed assuming we start execution (as usual) with a call to `main()`.

```
r(flag) {
    printf("Here !!!\n");
```

```
    }  
    q(flag) {  
        p(flag+1);  
    }  
    p(int flag) {  
        switch(flag) {  
            case 1: q(flag);  
            case 2: q(flag);  
            case 3: r(flag);  
        }  
    }  
    main() {  
        p(1);  
    }  
}
```

3. Consider the following C-like program that allows subprograms to nest. Show the sequence of frames, with static links, on the stack when `r(16)` is executed assuming we start execution (as usual) with a call to `main()`. Explain how the values of `a`, `b` and `c` are accessed in `r`'s `print` statement.

```
    p(int a) {  
        q(int b) {  
            r(int c) {  
                print(a+b+c);  
            }  
            r(b+3);  
        }  
    }  
}
```

```

    }
    s(int d){
        q(d+2);
    }
    s(a+1);
}
main(){
    p(10);
}

```

4. Reconsider the C-like program shown in Exercise 3, this time assuming display registers are used to access frames (rather than static links). Explain how the values of `a`, `b` and `c` are accessed in `r`'s `print` statement.
5. Consider the following C function. Show the content and structure of `f`'s frame. Explain how the offsets of `f`'s local variables are determined.

```

int f(int a, int b[]){
    int i = 0, sum = 0;
    while (i < 100){
        int val = b[i]+a;
        if (b[i]>b[i+1]) {
            int swap = b[i];
            b[i] = b[i+1];
            b[i+1] = swap;
        } else {

```

```
        int avg = (b[i]+b[i+1])/2;
        b[i] = b[i+1] = avg; }
    sum += val;
    i++;
}
return sum;
}
```

6. Although the first release of Java did not allow classes to nest, subsequent releases did. This introduced problems of nested access to objects, similar to those found when subprograms are allowed to nest. Consider the following Java class definition.

```
class Test {
    class Local {
        int b;
        int v(){return a+b;}
        Local(int val){b=val;}
    }
    int a = 456;
    void m(){
        Local temp = new Local(123);
        int c = temp.v();
    }
}
```

Note that method `v()` of class `Local` has access to field `a` of class `Test` as well as field `b` of class `Local`. However, when `temp.v()` is called it is given a direct reference only to `temp`. Suggest a variant of static links that can be used to implement nested classes so that access to all visible objects is provided.

7. Assume we organize a heap using reference counts. What operations must be done when a pointer to a heap object is assigned? What operations must be done when a scope is opened and closed?
8. Some languages, including C and C++, contain an operation that creates a pointer to a data object. That is, `p = &x` takes the address of object `x`, whose type is `t`, and assigns it to `p`, whose type is `t*`.

How is management of the run-time stack complicated if it is possible to create pointers to arbitrary data objects in frames? What restrictions on the creation and copying of pointers to data objects suffice to guarantee the integrity of the run-time stack?

9. Consider a heap allocation strategy we shall term *worst fit*. Unlike best fit, which allocates a heap request from the free space block that is closest to the requested size, worst fit allocates a heap request from the largest available free space block. What are the advantages and disadvantages of worst fit as compared with the best fit, first fit, and next fit heap allocation strategies?
10. The performance of complex algorithms is often evaluated by simulating their behavior. Create a program that simulates a random sequence of heap allocations and deallocations. Use it to compare the average number of iterations

that the best fit, first fit, and next fit heap allocation techniques require to find and allocate space for a heap object.

11. In a strongly typed language like Java all variables and fields have a fixed type known at compile-time. What run-time data structures are needed in Java to implement the mark phase of a mark-sweep garbage collector in which all accessible (“live”) heap objects are marked?
12. The second phase of a mark-sweep garbage collector is the sweep phase, in which all unmarked heap objects are returned to the free-space list.

Detail the actions needed to step through the heap, examining each object and identifying those that have been not been marked (and hence are garbage).

13. In a language like C or C++ (without unions), the marking phase of a mark-sweep garbage collector is complicated by the fact that pointers to active heap objects may reference data within an object rather than the object itself. For example, the sole pointer to an array may be to an internal element, or the sole pointer to a class object may be a pointer to one of the object’s fields.

How must your solution to Exercise 11 be modified if pointers to data within an object are allowed?

14. One of the attractive aspects of conservative garbage collection is its simplicity. We need not store detailed information on what global, local and heap variables are pointers. Rather, any word that *might* be a heap pointer is treated as if *is* a pointer.

What criteria would you use to decide if a given word in memory is possibly a pointer? How would you adapt your answer to Exercise 13 to handle what appear to be pointers to data within a heap object?

15. One of the most attractive aspects of copying garbage collectors is that collecting garbage actually costs nothing since only live data objects are identified and moved. Assuming that the total amount of heap space live at any point is constant, show that the average cost of garbage collection (per heap object allocated) can be made arbitrarily cheap by simply by increasing the memory size allocated to the heap.
16. Copying garbage collection can be improved by identifying long-lived heap objects and allocating them in an area of the heap that is not collected.

What compile-time analyses can be done to identify heap objects that will be long-lived? At run-time, how can we efficiently estimate the “age” of a heap object (so that long-lived heap objects can be specially treated)?

17. An unattractive aspect of both mark-sweep and copying garbage collects is that they are batch-oriented. That is, they assume that periodically a computation can be stopped while garbage is identified and collected. In interactive or real-time programs, pauses can be quite undesirable. An attractive alternative is *concurrent garbage collection* in which a garbage collection process runs concurrently with a program.

Consider both mark-sweep and copying garbage collectors. What phases of each can be run concurrently while a program is executing (that is, while the

program is changing pointers and allocating heap objects)? What changes to garbage collection algorithms can facilitate concurrent garbage collection?

18. Assume we are compiling a language like Pascal or Ada that allows nested procedures. Further, assume we are in procedure P , at nesting level m , and wish to access variable v declared in Procedure Q , which is at nesting level n . If v is at offset of in Q 's frame, and if static links are always stored at offset sl in a frame, what expression must be evaluated to access v from procedure P using static links?

12

Simple Code Generation

In this chapter we will begin to explore the final phase of compilation—code generation. At this point the compiler has already created an AST and type-checked it, so we know we have a valid source program. Now the compiler will generate machine-level instructions and data that correctly implement the semantics of the AST.

Up to now, our compiler has been completely target-machine independent. That is, nothing in the design of our scanner, parser and type-checker depends on exactly which computer we will be using. Code generation, of course, *does* depend on target-machine details. Here we'll need to know the range of instructions and addressing modes available, the format of data types, the range of operating system services that are provided, etc. Using this knowledge, the code generator will

translate each AST into an **executable form** appropriate for a particular target machine.

12.1 Assembly Language and Binary Formats

We'll first need to choose the exact form that we'll use for the code that we generate. One form, commonly used in student projects and experimental compilers, is **assembly language**. We simply generate a text file containing assembler instructions and directives. This file is then assembled, linked and loaded to produce an executable program.

Generating assembly language frees us from a lot of low-level machine-specific details. We need not worry about the exact bit patterns used to represent integers, floats and instructions. We can use symbolic labels, so we need not worry about tracking the exact numeric values of addresses. Assembly language is easy to read, simplifying the debugging of our code generator.

The disadvantage of generating assembly language is that an additional assembly phase is needed to put our program into a form suitable for execution. Assemblers can be rather slow and cumbersome. Hence, rather than generating assembly language, many production quality compilers (like the Java, C and C++ compilers you use) generate code in a **binary format** that is very close to that expected by the target machine.

To support independent compilation of program components, compilers usually produce a **relocatable object file** (a “.o” or “.class” file) that can be linked with other object files and library routines to produce an executable program.

Each instruction or data value is translated into its binary format. For integers this is typically the two's complement representation. For floats, the IEEE floating point standard is widely used. The format of instructions is highly machine-specific. Typically, the leftmost byte represents the operation, with remaining bytes specifying registers, offsets, immediate values or addresses.

Each instruction or data value is placed at a fixed offset within a **section**. When object files are linked together, all members of a section are assigned adjacent memory addresses in the order of their offsets. A section might represent the translated body of a subroutine, or the global variables of a “.c” file.

When references to the address of an instruction or data value are found, they are of the form “section plus offset.” Since we don't yet know exactly where in memory a section will be placed, these addresses are marked as **relocatable**. This means that when the linker is run, these address will be *automatically* adjusted to reflect the address assigned to each section.

An object file may reference labels defined in other object files (e.g., `printf`). These are **external references**. They are marked for relocation by the value of the external label, when the linker determines the actual value of the label. Similarly, labels of subroutines and exported global variables may be marked as **external definitions**. Their values may be referenced by other object files.

While this object file mechanism may seem complicated (and in many ways *it is*), it does allow us to build programs from separately compiled pieces, including standard library routines. Not all the object files need be created by the same compiler; they need not even be programed in the same source language. The linker

will relocate addresses and resolve cross-file references. Some compilers, like those for ML and Java, perform an **inter-module dependency analysis** on separately compiled modules to check interface and type consistency.

In Java, execution begins with a single class file. Its fields are initialized and its methods are loaded. In the interests of security, methods may be **verified** to guarantee that they do not violate type security or perform illegal operations. Other classes that are accessed may be **dynamically linked** and loaded as they are referenced, thereby building a program class by class.

In some cases all we want to do is translate and execute a program immediately. The generation and linkage of object files would be unnecessarily complex. An alternative is to generate **absolute machine code**. This comprises machine-level instructions and data with all addresses fully resolved. As instructions and data are generated, they are given fixed and completely resolved addresses, typically within a large block of memory preallocated by the compiler. If an address's value isn't known yet (because it represents an instruction or data value not yet generated), the address is **backpatched**. That is, the location where the address is needed is stored, and the address is "patched in" once it becomes known. For example, if we translate `goto L` before `L`'s address is known, we provisionally generate a jump instruction to location 0 (`jmp 0`) with the understanding that address 0 will be corrected later. When `L` is encountered, its address becomes known (e.g., 1000) and `jmp 0` is updated to `jmp 1000`.

Library routines are loaded (in fully translated form) as necessary. After all instructions and data are translated into binary form and loaded into memory,

execution begins. Files like `stdin` and `stdout` are opened, and control is passed to the entry point address (usually the first instruction of `main`).

12.2 Translating ASTs

Intermediate representations of programs, like ASTs, are oriented toward the source language being translated. They reflect the source constructs of a language, like loops and arrays and procedures, without specifically stating how these constructs will be translated into executable form. The first step of code generation is therefore **translation**, in which the meaning of a construct is reduced to a number of simple, easily implemented steps. Thus a binary expression that applies an operator to left and right operands (e.g., `a[i] + b[j]`) might be translated by the rule:

Translate the left operand, then translate the right operand,
then translate the operator

After a construct is translated, it becomes necessary to choose the particular machine language instructions that will be used to implement the construct. This is called **instruction selection**. Instruction selection is highly target-machine dependent. The same construct, translated the same way, may be implemented very differently on different computers. Even on the same machine, we often have a choice of possible implementations. For example, to add one to a value, we might load the literal one into a register and do a register to register add. Alternatively, we might do an add immediate or increment instruction, avoiding an unnecessary load of the constant operand.

12.3 Code Generation for ASTs

Let us first consider the steps needed to design and implement a code generator for an AST node. Consider the **conditional expression**, `if exp1 then exp2 else exp3`. (Conditional expressions are represented in Java, C and C++ in the form `exp1?exp2:exp3`).

We first consider the **semantics** of the expression—what does it *mean*? For conditional expressions, we must evaluate the first expression. If its value is non-zero, the second expression is evaluated and returned as the result of the conditional expression. Otherwise, the third expression is evaluated and returned as the value of the conditional expression.

Since our code generator will be translating an AST, the exact syntax of the conditional expression is no longer visible. The code generator sees an AST like the one shown in Figure 12.1. Rectangles represent single AST nodes; triangles represent AST subtrees. The type of the node is listed first, followed by the fields it contains.

The type checker has already been run, so we know the types are all correct. Type rules of the source language may require conversion of some operands. In such cases explicit conversion operators are inserted as needed (e.g., `a<b?1:1.5` becomes `a<b?((float)1):1.5`).

We first outline the translation we want. A portion of the translation will be done at `ConditionalExprNode`; the remainder will be done by *recursively visiting*

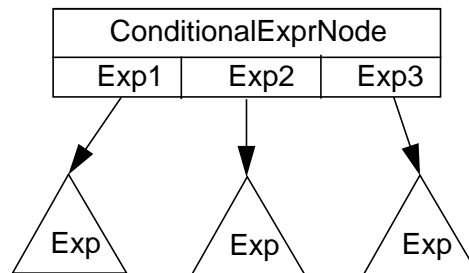


Figure 12.1 Abstract Syntax Tree for a Conditional Expression

its subtrees. Each node's translation will mesh together to form the complete translation.

In choosing a translation it is vital that we fully understand the semantics of the construct. We can study a language reference manual, or consult with a language designer, or examine the translations produced by some reference compiler.

The code we choose will operate as follows:

1. Evaluate `Exp1`.
2. If `Exp1` is zero go to 5.
3. Evaluate `ans ← Exp2`.
4. Go to 6.
5. Evaluate `ans ← Exp3`.
6. Use the value computed in `ans` as the result of the conditional expression.

We have not yet chosen the exact instructions we will generate—only the specific steps to be performed by the conditional expression. Note that alternative translations may be possible, but we must be sure that any translation we choose

is *correct*. Thus evaluating *both* Exp2 and Exp3 would not only be inefficient, it would be *wrong*. (Consider `if a==0 then 1 else 1/a`).

Next we choose the instructions we wish to generate. A given translation may be implemented by various instruction sequences. We want the fastest and most compact instruction sequence possible. Thus we won't use three instructions when two will do, and won't use slower instructions (like a floating-point multiply) when a faster instruction (like a floating-point add) might be used instead.

At this stage it is important that we be very familiar with the architecture of the machine for which we are generating code. As we design and choose the code we will use to implement a particular construct, it is a good idea actually to try the instruction sequences that have been chosen to verify that they will work properly. Once we're sure that the code we've chosen is viable, we can include it in our code generator.

In the examples in this chapter we'll use the Java Virtual Machine (JVM) instruction set [Lindholm and Yellin 1997]. This architecture is clean and easy to use and yet is implemented on virtually all current architectures. We'll also use Jasmin [Meyer and Downing 1997], a symbolic assembler for the JVM.

In Chapter 15 we extend our discussion of code generation to include issues of instruction selection, register allocation and code scheduling for mainstream architectures like the MIPS, Sparc and PowerPC. Thus the JVM instructions we generate in this chapter may be viewed as a convenient platform-independent intermediate form that can be expanded, if necessary, into a particular target machine form.

Before we can translate a conditional expression, we must know how to translate the variables, constants and expressions that may appear within it. Thus we will first study how to translate declarations and uses of simple scalar variables and constants. Then we'll consider expressions, including conditional expressions. Finally we'll look at assignment operations. More complex data structures and statements will be discussed in succeeding chapters.

12.4 Declaration of Scalar Variables and Constants

We will begin our discussion of translation with the declaration of simple scalar variables. Variables may be global or local. Global variables are those declared outside any subprogram or method as well as static variables and fields. Non-static variables declared within a subprogram or method are considered local. We'll consider globals first.

A variable declaration consists of an identifier, its type, and an optional constant-valued initialization, as shown in Figure 12.2.

An `IdentifierNode` will contain an `Address` field (set by the type-checker) that references an `AddressNode`. This field records how the value the identifier represents is to be accessed. Identifiers may access the values they denote in many ways: as global values (statically allocated), as local values (allocated within a stack frame), as register values (held only in a register), as stack values (stored on the run-time stack), as literal values, or as indirect values (a pointer to the actual value). The `AccessMode` field in `AddressNode` records how an identifier (or any other data object) is to be accessed at run-time.

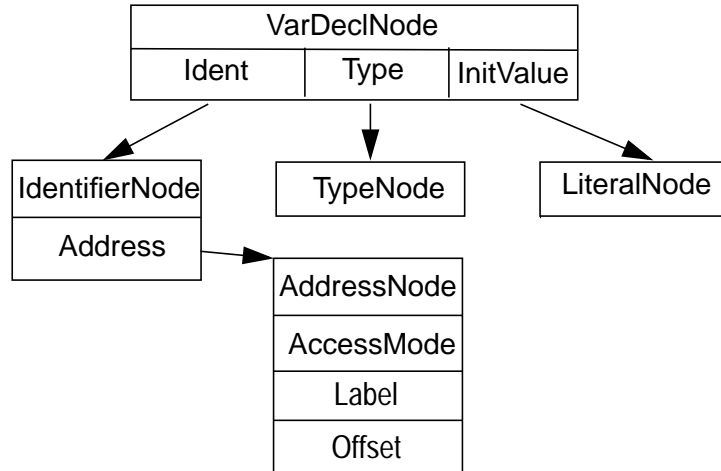


Figure 12.2 Abstract Syntax Tree for Variable Declarations

Within a program, all uses of a given identifier will share the same **IdentifierNode**, so all information added to an AST node during declaration processing will be visible when uses of the identifier are processed.

To generate code, we'll use a variety of "code generation" subroutines, each prefixed with "Gen." A code generation subroutine generates the machine level instructions needed to implement a particular elementary operation (like loading a value from memory or conditionally branching to a label). With care, almost all machine-specific operations can be compartmentalized within these subroutines.

In our examples, we'll assume that code generation subroutines are generating JVM instructions. However, it is a simple matter to recode these subroutines to generate code for other computers, like the Sparc, or x86 or PowerPC. Our discussion is in no sense limited to just Java and the JVM.

We'll also assume that our code generation subroutines generate assembly language. Each subroutine will append one or more lines into an output file. When code generation is complete, the file will contain a complete assembly language program, ready for processing using Jasmin or some other assembler.

In translating variable declarations, we will use the code generation subroutine `GenGlobalDecl(Label, Type, InitialVal)`. `Label` is a unique label for the declaration. It may be generated using the subroutine `CreateUniqueLabel()` that creates a unique label each time it is called (e.g., `Lab001`, `Lab002`, etc.). Alternatively, the label may be derived from the identifier's name. Be careful however; many assemblers treat certain names as reserved (for example, the names of op codes). To avoid an unintentional clash, you may need to suffix identifier names with a special symbol (such as \$).

`Type` is used to determine the amount of memory to reserve. Thus an `int` will reserve a word (4 bytes), a `char` a single byte or halfword, etc.

`InitialVal` is the literal value the global location will be initialized to. It is obtained from the `LiteralValue` field of the `LiteralNode`. If no explicit initial value is provided, `InitialVal` is null and a default initial value (or none at all) is used.

For example, if we are generating Jasmin JVM instructions, `GenGlobalDecl(Label, Type, InitialVal)` might generate

```
.field public static Label TypeCode = Init
```

In the JVM globals are implemented as static fields. `TypeCode` is a type code derived from the value of `Type`. `Init` is the initial value specified by `InitialVal`. `Label` is an assembly language label specified by `Label`.

Recall that variables local to a subprogram or method are considered to be part of a stack frame that is pushed whenever the procedure is called (see section 11.2). This means we can't generate individual labels for local variables—they don't have a fixed address. Rather, when a local declaration is processed during code generation the variable is assigned a fixed offset within the frame. If necessary, offsets are adjusted to guarantee that data is properly aligned. Thus a local integer variable will always have an offset that is word aligned, even if it immediately follows a character variable.

The size of the frame is increased to make room for the newly declared variable. After all declarations are processed, the final size of the frame is saved as part of the routine's type information (this size will be needed when the routine is called). A local variable that is initialized is handled as an uninitialized declaration followed by an assignment statement. That is, explicit run-time initialization instructions are needed each time a procedure is called and its frame is pushed. That is why C and C++ do not automatically initialize local variables.

In the JVM, local variables are easy to implement. All local variables, including parameters, are given a **variable index** in the current frame. This index, in units of words, is incremented by one for all types that require one word or less, and is incremented by two for data that require a doubleword.

We'll use the subroutine `GenLocalDecl(Type)` to process a local declaration. This routine will return the offset (or variable index) assigned to the local variable (based on its `Type`).

Each AST node will contain a member function `CodeGen()` that generates the code corresponding to it. This routine will generate assembly language instructions by calling code generation subroutines. The code generator for a `VarDeclNode` is shown in Figure 12.3. (Implementation of assignments is discussed in section Section 12.9).

```

VarDeclNode.CodeGen( )
1.  if Ident.Address.AccessMode = Global
2.      then Ident.Address.Label ← CreateUniqueLabel()
3.          GenGlobalDecl(Ident.Address.Label, Type, InitValue)
4.      else Ident.Address.Offset ← GenLocalDecl(Type)
           if InitValue ≠ null
5.          then /* Generate code to store InitValue in Ident */

```

Figure 12.3 Code Generation for Variable Declarations

Note that either `Label` or `Offset` in `AddressNode` is set during code generation for variable declarations. These fields are used when uses of the identifier are translated.

As mentioned earlier, one of the problems in using global (32 or 64 bit) addresses is that they often “don’t fit” within a single instruction. This means that referencing a global variable by its label may generate several instructions rather than one. To improve efficiency, many compilers access globals using an offset relative to a **global pointer** that points to the global data area (see Section 11.1).

This approach is easy to accommodate. As globals are declared their `Offset` field is set along with their `Label` field. The `Offset` can be used to load or store globals in one instruction. The `Label` field can be used to access global variables in

other compilation units or if the global data area is so large that not all variable addresses can be represented as an offset relative to a single register.

12.4.1 Handling Constant Declarations and Literals

Constant declarations can be handled almost exactly the same as initialized variables. That is, we allocate space for the constant and generate code to initialize it. Since constants may not be changed (the type-checker enforces this), the location allocated to the constant may be referenced whenever the constant's value is needed.

A few optimizations are commonly applied. For named constants whose value fits within the range of immediate operands, we need not allocate a memory location. Rather, we store the explicit numeric value of the constant and “fill it in” as an immediate operand.

Locally declared constants may be treated like static variables and allocated a global location. This eliminates the need to reinitialize the constant each time the routine that contains it is called.

Small integer literals are accessed when possible as immediate operands. Larger integer literals, as well as floating literals, are handled like named constants. They are allocated a global data location and the value of the literal is fetched from its memory location. Globally allocated literal values may be

accessed through the global pointer if their addresses are too large to fit in a single instruction.

Some compilers keep a list of literals that have been placed in memory so that all occurrences of the same literal share the same memory location. Other compilers place literals in memory each time they are encountered, simplifying compilation a bit at the expense of possibly wasting a few words of memory.

12.5 Translating Simple Expressions

We now consider the translation of simple expressions, involving operands that are simple scalar variables, constants or literals. Our solution generalizes nicely to expressions involving components of arrays or fields in classes or structures, or the results of function calls.

Many architectures require that an operand be loaded into a register before it can be moved or manipulated. The JVM requires that an operand be placed on the run-time stack before it is manipulated.

We will therefore expect that calling `CodeGen` in an AST that represents an expression will have the effect of generating code to compute the value of that expression into a register or onto the stack. This may involve simply loading a variable from memory, or it may involve a complex computation implemented using many instructions. In any event, we'll expect to have an expression's value in a register or on the stack after it is translated.

12.5.1 Temporary Management

Values computed in one part of an expression must be communicated to other parts of the expression. For example, in $exp1+exp2$, $exp1$ must be computed and then held while $exp2$ is computed. Partial or intermediate values computed within an expression or statement are called **temporary values** (because they are needed only temporarily). Temporaries are normally kept on the stack or in registers, though storage temporaries are sometimes used.

To allocate and free temporaries, we'll create two code generation subroutines, `GetTemp(Type)` and `FreeTemp(Address1, Address2, ...)`. `GetTemp`, which will return an `AddressNode`, will allocate a temporary location that can hold an object of the specified type.

On a register-oriented machine, `GetTemp` will normally return an integer or floating point register. It will choose the register from a pool of registers reserved to hold operands (see Section 15.3.1).

On a stack-oriented machine like the JVM, `GetTemp` will do almost nothing. This is because results are almost always computed directly onto the top of the stack. Therefore `GetTemp` will simply indicate in the address it returns that the stack is to be used.

For more complex objects like arrays or strings, `GetTemp` may allocate memory locations (in the current frame or heap) and return a pointer to the locations allocated.

`FreeTemp`, called when temporary locations are no longer needed, frees the locations at the specified addresses. Registers are returned to the pool of available registers so that they may be reused in future calls to `GetTemp`.

For stack-allocated temporaries a pop may need to be generated. For the JVM, operations almost always pop operands after they are used, so `FreeTemp` need not pop the stack. Temporaries allocated in memory may be released, and their memory locations may be later reused as necessary.

Now we know how to allocate temporaries, but we still need a way of communicating temporary information between AST nodes. We will add a field called `Result` to AST nodes. `Result` contains an `AddressNode` that describes the temporary that will contain the value the AST node will compute. If the AST computes no value into a temporary, the value of `Result` is ignored.

Who sets the value of `Result`? There are two possibilities. A parent can set the value, in effect directing a node to compute its value into the location its parent has chosen. Alternatively, a node can set the value itself, informing its parent of the temporary it has chosen to compute its value into.

In most cases the parent doesn't care what temporary is used. It will set `Result` to null to indicate that it will accept any temporary. However sometimes it is useful to force a value into a particular temporary, particularly when registers are used (this is called **register targeting**).

Reconsider our conditional expression example. If we let each node choose its own result register, `exp2` and `exp3` *won't* use the same register. And yet in this case, that's exactly what we want, since `exp2` and `exp3` are never both evaluated. If the `ConditionalExprNode` sets `Result`, it can direct both subtrees to use the same register. In the JVM, results almost always are placed on the stack, so targeting is automatic (and trivial).

12.5.2 Translating Simple Variables, Constants and Literals

We'll now consider how to generate code for AST nodes corresponding to variables, constants and literals. Variables and constants are both represented by `IdentifierNodes`. Variables and constants may be local or global and of integer or floating type. Non-scalar variables and constants, components of arrays, members of classes, and parameters will be handled in later chapters.

To translate scalar variables and constants, we will use three code generation subroutines:

```
GenLoadGlobal(Result, Type, Label),
```

```
GenLoadLocal(Result, Type, Offset),
```

```
GenLoadLiteral(Result, Type, Value).
```

`GenLoadGlobal` will generate a load of a global variable using the label and type specified. If `Result` is non-null, it will place the variable in `Result`. Otherwise, `GenLoadGlobal` will call `GetTemp` and return the temporary location it has loaded.

Similarly, `GenLoadLocal` will generate a load of a local variable using the offset and type specified. If `Result` is non-null, it will place the variable in `Result`. Otherwise, `GenLoadLocal` will call `GetTemp` and return the temporary location it has loaded.

`GenLoadLiteral` will generate a load of a literal value using the type specified. If `Result` is non-null, it will place the literal in `Result`. Otherwise, `GenLoadLiteral` will call `GetTemp` and return the temporary location it has loaded.

If a constant represents a literal value (e.g., `const float pi = 3.14159`) and `Result` is not set by a parent AST node, our code generator will set `Result` to indicate a literal value. This allows operators that have literal operands to produce better code, perhaps generating an immediate form instruction, or **folding** an expression involving only literals into a literal result. The code generator for `IdentifierNodes` is shown in Figure 12.4.

```
IdentifierNode.CodeGen( )
1.  if Address.AccessMode = Global
2.      then Result ← GenLoadGlobal(Result, Type, Address.Label)
3.  elsif Address.AccessMode = Local
4.      then Result ← GenLoadLocal(Result, Type, Address.Offset)
5.  elsif Address.AccessMode = Literal
6.      then if Result ≠ null
7.          then Result ← GenLoadLiteral(Result, Type, Address.Value)
8.          else Result ← AddressNode(Literal, Address.Value)
```

Figure 12.4 Code Generation for Simple Variables and Constants

AST nodes representing literals are handled in much the same manner as `IdentifierNodes`. If `Result` is set by a parent node, `GenLoadLiteral` is used to load the literal value into the request result location. Otherwise, the `Result` field is set to represent the fact that it is a literal value (`AccessMode = Literal`). A load into a register or onto the stack is delayed to allow optimizations in parent nodes.

12.6 Translating Predefined Operators

We now consider the translation of simple expressions—those involving the standard predefined operators and variable, constant or literal operands. In most cases these operators are easy to translate. Most operators have a machine instruction that directly implements them. Arithmetic operations usually require operands be on the stack or in registers, which fits our translation scheme well. We translate operands first and then apply the machine operation that corresponds to the operator being translated. The result is placed in a temporary (**Result**) just as expected.

A few complication can arise. Some machine instructions allow an immediate operand. We'd like to exploit that possibility when appropriate (rather than loading a constant operand unnecessarily onto the stack or into a register). If both operands are literals, we'd like to be able to simply compute the result at compile-time. (This is a common optimization called **folding**.) For example, $a = 100 - 1$ should be compiled as if it were $a = 99$, with no run-time code generated to do the subtraction.

Care must also be taken to free temporaries properly after they are used as operands (lest we “lose” a register for the rest of the compilation).

We'll use a code generation subroutine `GenAdd(Result, Type, Left, Right)`. `GenAdd` will generate code to add the operands described by `Left` and `Right`. These may be temporaries holding an already-evaluated operand or literals. If literals are involved, `GenAdd` may be able to generate an immediate instruction or it may be able to fold the operation into a literal result (generating no code at all).

`GenAdd` will use `Type` to determine the kind of addition to do (integer or floating, single or double length, etc.). It won't have to worry about "mixed mode" operations (e.g., an integer plus a float) as the type checker will have already inserted type conversion nodes into the AST as necessary. If `Result` is non-null, `GenAdd` will place the sum in `Result`. Otherwise, `GenAdd` will call `GetTemp` and return the temporary location into which it has computed the sum.

Since addition is commutative ($a+b \equiv b+a$) we can easily extend `GenAdd` to handle cases in which it is beneficial to swap the left and right operands.

The code generator for `PlusNodes` is shown in Figure 12.5.

```
PlusNode.CodeGen( )
1. LeftOperand.CodeGen()
2. RightOperand.CodeGen()
3. Result ← GenAdd(Result, Type,
                   LeftOperand.Result, RightOperand.Result)
4. FreeTemp(LeftOperand.Result, RightOperand.Result)
```

Figure 12.5 Code Generation for + Operator

As an example, consider the expression $A+B+3$ where `A` is a global integer variable with a label of `L1` in class `C` and `B` is a local integer variable assigned a local index (an offset within the frame) of 4. The JVM code that is generated is illustrated in Figure 12.6.

JVM Code	Comments	Generated By
<code>getstatic C/L1 I</code>	Push static integer field onto stack	<code>GenLoadGlobal</code>
<code>iload 4</code>	Push integer local 4 onto stack	<code>GenLoadLocal</code>
<code>iadd</code>	Add top two stack locations	<code>GenAdd</code>
<code>iconst_3</code>	Push integer literal 3 onto stack	<code>GenAdd</code>
<code>iadd</code>	Add top two stack locations	<code>GenAdd</code>

Figure 12.6 JVM Code Generated for $A+B+3$

Other arithmetic and logical operators are handled in a similar manner. Figure 12.7 lists a variety of common integer, relational and logical operators and their corresponding JVM op codes. Since unary + is the identity operator, it is implemented by doing nothing ($+b \equiv b$). Further, since $!b \equiv b==0$, ! is implemented using the JVM op code for comparison with zero. There are corresponding instructions for long integer operands and single- and double-length floating operands.

Binary Operator	+	-	*	/	&	
JVM Op Code	iadd	isub	imul	idiv	iand	ior
Binary Operator	^	<<	>>	%		
JVM Op Code	ixor	ishl	ishr	irem		
Binary Operator	<	>	<=	>=	==	!=
JVM Op Code	if_icmplt	if_icmplt	if_icmplt	if_icmplt	if_icmplt	if_icmplt
Unary Operator	+	-	!	~		
JVM Op Code		ineg	ifeq	ixor		

Figure 12.7 Common Operators and Corresponding JVM OP Codes

Relational operators, like == and >=, are a bit more clumsy to implement. The JVM does not provide instructions that directly compare two operands to produce a boolean result. Rather, two operands are compared, resulting in a conditional branch to a given label. Similarly, many machines, including the Sparc, Motorola MC680x0, Intel x86, IBM RS6000 and PowerPC, use **condition codes**. After a comparison operation, the result is stored in one or more **condition code bits** held

in the processor's status word. This condition code must then be extracted or tested using a conditional branch.

Consider the expression $A < B$ where A and B are local integer variables with frame offsets 2 and 3. On the JVM we might generate.

```
    iload      2  ; Push local #2 (A) onto the stack
    iload      3  ; Push local #3 (B) onto the stack
    if_icmplt  L1 ; Goto L1 if A < B
    iconst_0    ; Push 0 (false) onto the stack
    goto       L2 ; Skip around next instruction
L1: iconst_1    ; Push 1 (true) onto the stack
L2:
```

This instruction sequence uses six instructions, whereas most other operations require only three (push both operands, then apply an operator). Fortunately in the common case in which a relational expression controls a conditional or looping statement, better code is possible.

12.6.1 Short-Circuit and Conditional Evaluation

We have not yet discussed the $\&\&$ and $\|\|$ operators used in C, C++ and Java. That's because they are special. Unlike most binary operators, which evaluate both their operands and then perform their operation, these operators work in "short circuit" mode. This is, if the left operand is sufficient to determine the result of the operation, the right operand *isn't evaluated*. In particular $a\&\&b$ is defined as if a then b else $false$. Similarly $a\|\|b$ is defined as if a then

true else b. The conditional evaluation of the second operand isn't just an optimization—it's essential for correctness. Thus in $(a \neq 0) \ \&\& \ (b/a > 100)$ we would perform a division by zero if the right operand were evaluated when $a = 0$.

Since we've defined $\&\&$ and $\|\|$ in terms of conditional expressions, let's complete our translation of conditional expressions, as shown in Figure 12.8. `GenLabel(Label)` generates a definition of `Label` in the generated program. `GenBranchIfZero(Operand, Label)` generates a conditional branch to `Label` if `Operand` is zero (false). `GenGoTo(Label)` generates a goto to `Label`.

```
ConditionalExprNode.CodeGen( )
1.  Result ← Exp2.Result ← Exp3.Result ← GetTemp()
2.  Exp1.CodeGen()
3.  FalseLabel ← CreateUniqueLabel()
4.  GenBranchIfZero(Exp1.Result, FalseLabel)
5.  FreeTemp(Exp1.Result)
6.  Exp2.CodeGen()
7.  OutLabel ← CreateUniqueLabel()
8.  GenGoTo(OutLabel)
9.  GenLabel(FalseLabel)
10. Exp3.CodeGen()
11. GenLabel(OutLabel)
```

Figure 12.8 Code Generation for Conditional Expressions

Our code generator follows the translation pattern we selected in Section 12.3. First `exp1` is evaluated. We then generate a “branch if equal zero” instruction that tests the value of `exp1`. If `exp1` evaluates to zero, which represents false, we branch to the `FalseLabel`.

Otherwise, we “fall through.” `exp2` is evaluated into a temporary. Then we branch unconditionally to `OutLabel`, the exit point for this construct. Next `False-`

Label is defined. At this point `exp3` is evaluated, into the same location as `exp2`. Then `OutLabel` is defined, to mark the end of the conditional expression.

As an example, consider `if B then A else 1`. We generate the following JVM instructions, assuming `A` has a frame index of 1 and `B` has a frame index of 2.

```

    iload      2 ; Push local #2 (B) onto the stack
    ifeq      L1 ; Goto L1 if B is 0 (false)
    iload      1 ; Push local #1 (A) onto the stack
    goto      L2 ; Skip around next instruction
L1:  iconst_1   ; Push 1 onto the stack
L2:

```

12.6.2 Jump Code Evaluation

The code we generated for `if B then A else 1` is simple and efficient. For a similar expression, `if (F==G) then A else 1` (where `F` and `G` are local variables of type integer), we'd generate

```

    iload      4 ; Push local #4 (F) onto the stack
    iload      5 ; Push local #5 (G) onto the stack
    if_icmpeq  L1 ; Goto L1 if F == G
    iconst_0   ; Push 0 (false) onto the stack
    goto      L2 ; Skip around next instruction
L1:  iconst_1   ; Push 1 (true) onto the stack
L2:  ifeq      L3 ; Goto L3 if F == G is 0 (false)
    iload      1 ; Push local #1 (A) onto the stack

```

```

        goto      L4 ; Skip around next instruction
L3:  iconst_1    ; Push 1 onto the stack
L4:

```

This code is a bit clumsy. We generate a conditional branch to set the value of $F==G$ just so that we can conditionally branch on that value. Any architecture that uses condition codes will have a similar problem.

A moment's reflection shows that we rarely actually *want* the value of a relational or logical expression. Rather, we usually only want to do a conditional branch based on the expression's value, in the context of a conditional or looping statement.

Jump code is an alternative representation of boolean values. Rather than placing a boolean value directly in a register or on the stack, we generate a conditional branch to either a **true label** or a **false label**. These labels are defined at the places where we wish execution to proceed once the boolean expression's value is known.

Returning to our previous example, we can generate $F==G$ in jump code form as

```

        iload     4 ; Push local #4 (F) onto the stack
        iload     5 ; Push local #5 (G) onto the stack
        if_icmpne L1 ; Goto L1 if F != G

```

The label L1 is the “false label.” We branch to it if the expression $F == G$ is false; otherwise, we fall through, executing the code that follows if the expression

is true. We can then generate the rest of the expression, defining L1 at the point where the else expression is to be computed:

```
        iload        1 ; Push local #1 (A) onto the stack
        goto        L2 ; Skip around next instruction
L1:  iconst_1       ; Push 1 onto the stack
L2:
```

This instruction sequence is significantly shorter (and faster) than our original translation.

Jump code comes in two forms, `JumpIfTrue` and `JumpIfFalse`. In `JumpIfTrue` form, the code sequence does a conditional jump (branch) if the expression is true, and “falls through” if the expression is false. Analogously, in `JumpIfFalse` form, the code sequence does a conditional jump (branch) if the expression is false, and “falls through” if the expression is true. We have two forms because different contexts prefer one or the other.

We will augment our definition of `AddressNodes` to include the `AccessMode` values `JumpIfTrue` and `JumpIfFalse`. The `Label` field of the `AddressNode` will be used to hold the label conditionally jumped to.

It is important to emphasize that even though jump code looks a bit unusual, it is just an alternative representation of boolean values. We can convert a boolean value (on the stack or in a register) to jump code by conditionally branching on its value to a true or false label. Similarly, we convert from jump code to an explicit boolean value, by defining the jump code’s true label at a load of 1 and the false label at a load of 0. These conversion routines are defined in Figure 12.9.

```
ConvertToJumpCode ( Operand, AccessMode, Label )
```

1. if AccessMode = JumpIfFalse
2. then GenBranchIfZero(Operand, Label)
3. else GenBranchIfNonZero(Operand, Label)
4. return AddressNode(AccessMode, Label)

```
ConvertFromJumpCode ( Target, AccessMode, Label )
```

1. if AccessMode = JumpIfFalse
2. then FirstValue \leftarrow 1
3. SecondValue \leftarrow 0
4. else FirstValue \leftarrow 0
5. SecondValue \leftarrow 1
6. Target \leftarrow GenLoadLiteral(Target, Integer, FirstValue)
7. SkipLabel \leftarrow CreateUniqueLabel()
8. GenGoTo(SkipLabel)
9. GenLabel(Label)
10. Target \leftarrow GenLoadLiteral(Target, Integer, SecondValue)
11. GenLabel(SkipLabel)
12. return Target

Figure 12.9 Routines to Convert To and From Jump Code

We can easily augment the code generators for `IdentifierNodes` and `LiteralNodes` to use `ConvertToJumpCode` if the `Result` field shows jump code is required.

An advantage of the jump code form is that it meshes nicely with the `&&` and `||` operators, which are already defined in terms of conditional branches. In particular if `exp1` and `exp2` are in jump code form, then we need generate *no further code* to evaluate `exp1&&exp2`.

To evaluate `&&`, we first translate `exp1` into `JumpIfFalse` form, followed by `exp2`. If `exp1` is false, we jump out of the whole expression. If `exp1` is true, we fall through to `exp2` and evaluate it. In this way, `exp2` is evaluated only when necessary (when `exp1` is true).

The code generator for `&&` is shown in Figure 12.10. Note that both `JumpIfFalse` and `JumpIfTrue` forms are handled, which allows a parent node to select the form more suitable for a given context. A non-jump code result (in a register or on the stack) can also be produced, using `ConvertFromJumpCode`.

```

ConditionalAndNode.CodeGen( )
1.  if Result.AccessMode = JumpIfFalse
2.      then Exp1.Result ← AddressNode(JumpIfFalse,Result.Label)
3.          Exp2.Result ← AddressNode(JumpIfFalse,Result.Label)
4.          Exp1.CodeGen()
5.          Exp2.CodeGen()
6.  elsif Result.AccessMode = JumpIfTrue
7.      then Exp1.Result ← AddressNode(JumpIfFalse, CreateUniqueLabel())
8.          Exp2.Result ← AddressNode(JumpIfTrue,Result.Label)
9.          Exp1.CodeGen()
10.         Exp2.CodeGen()
11.         GenLabel(Exp1.Result.Label)
12.     else
13.         Exp1.Result ← AddressNode(JumpIfFalse, CreateUniqueLabel())
14.         Exp2.Result ← AddressNode(JumpIfFalse,Exp1.Result.Label)
15.         Exp1.CodeGen()
16.         Exp2.CodeGen()
17.         Result ← ConvertFromJumpCode(Result, JumpIfFalse,
                                         Exp1.Result.Label)

```

Figure 12.10 Code Generation for Conditional And

Similarly, once `exp1` and `exp2` are in jump code form, `exp1 || exp2` is easy to evaluate. We first translate `exp1` into `JumpIfTrue` form, followed by `exp2`. If `exp1` is true, we jump out of the whole expression. If `exp1` is false, we fall through to `exp2` and evaluate it. In this way, `exp2` is evaluated only when necessary (when `exp1` is false).

The code generator for `||` is shown in Figure 12.11. Again, both `JumpIfFalse` and `JumpIfTrue` forms are handled, allowing a parent node to select the form more

suitable for a given context. A non-jump code result (in a register or on the stack) can also be produced.

```

ConditionalOrNode.CodeGen ( )
1.  if Result.AccessMode = JumpIfTrue
2.      then Exp1.Result ← AddressNode(JumpIfTrue,Result.Label)
3.          Exp2.Result ← AddressNode(JumpIfTrue,Result.Label)
4.          Exp1.CodeGen()
5.          Exp2.CodeGen()
6.  elsif Result.AccessMode = JumpIfFalse
7.      then Exp1.Result ← AddressNode(JumpIfTrue, CreateUniqueLabel())
8.          Exp2.Result ← AddressNode(JumpIfFalse,Result.Label)
9.          Exp1.CodeGen()
10.         Exp2.CodeGen()
11.         GenLabel(Exp1.Result.Label)
12.     else
13.         Exp1.Result ← AddressNode(JumpIfTrue, CreateUniqueLabel())
14.         Exp2.Result ← AddressNode(JumpIfTrue,Exp1.Result.Label)
15.         Exp1.CodeGen()
16.         Exp2.CodeGen()
17.         Result ← ConvertFromJumpCode (Result, JumpIfTrue,
                                         Exp1.Result.Label)

```

Figure 12.11 Code Generation for Conditional Or

The ! operator is also very simple once a value is in jump code form. To evaluate !exp in JumpIfTrue form, you simply translate exp in JumpIfFalse form. That is, jumping to location L when exp is false is equivalent to jumping to L when !exp is true. Analogously, to evaluate !exp in JumpIfFalse form, you simply translate exp in JumpIfTrue form.

As an example, let's consider $(A > 0) \parallel (B < 0 \ \&\& \ C == 10)$, where A, B and C are all local integers, with indices of 1, 2 and 3 respectively. We'll produce a JumpIfFalse translation, jumping to label F if the expression is false and falling through if the expression is true.

The code generators for relational operators can be easily modified to produce both kinds of jump code—we can either jump if the relation holds (`JumpIfTrue`) or jump if it doesn't hold (`JumpIfFalse`). We produce the following JVM code sequence which is compact and efficient.

```
    iload      1      ; Push local #1 (A) onto the stack
    ifgt      L1      ; Goto L1 if A > 0 is true
    iload      2      ; Push local #2 (B) onto the stack
    ifge      F       ; Goto F if B < 0 is false
    iload      3      ; Push local #3 (C) onto the stack
    bipush    10      ; Push a byte immediate value (10)
    if_icmpne F       ; Goto F if C != 10
```

L1:

First A is tested. If it is greater than zero, the expression must be true, so we go to the end of the expression (since we wish to branch if the expression is *false*). Otherwise, we continue evaluating the expression.

We next test B. If it is greater than or equal to zero, $B < 0$ is false, and so is the whole expression. We therefore branch to label F as required. Otherwise, we finally test C. If C is not equal to 10, the expression is false, so we branch to label F. If C is equal to 10, the expression is true, and we fall through to the code that follows.

As shown in Figure 12.12, conditional expression evaluation is simplified when jump code is used. This is because we no longer need to issue a conditional

branch after the control expression is evaluated. Rather, a conditional branch is an integral part of the jump code itself.

As an example, consider `if (A&&B) then 1 else 2`. Assuming A and B are locals, with indices of 1 and 2, we generate

```

        iload    1      ; Push local #1 (A) onto the stack
        ifeq    L1      ; Goto L1 if A is false
        iload    2      ; Push local #2 (B) onto the stack
        ifeq    L1      ; Goto L1 if B is false
        iconst_1      ; Push 1 onto the stack
        goto    L2      ; Skip around next instruction

L1: iconst_2      ; Push 2 onto the stack

L2:

```

`ConditionalExprNode.CodeGen()`

1. `Result ← Exp2.Result ← Exp3.Result ← GetTemp()`
2. `FalseLabel ← CreateUniqueLabel()`
3. `Exp1.Result ← AddressNode(JumpIfFalse, FalseLabel)`
4. `Exp1.CodeGen()`
5. `Exp2.CodeGen()`
6. `OutLabel ← CreateUniqueLabel()`
7. `GenGoTo(OutLabel)`
8. `GenLabel(FalseLabel)`
9. `Exp3.CodeGen()`
10. `GenLabel(OutLabel)`

Figure 12.12 Code Generation for Conditional Expressions Using Jump Code

12.7 Pointer and Reference Manipulations

In C and C++ the unary `&` and `*` operators create and dereference pointers. For example,

```
P = &I; // P now points to I's memory location
J = *P; // Assign value P points to (I) to J
```

Java does not allow explicit pointer manipulation, so `&` and `*` are not available.

In most architectures, `&` and `*` are easy to implement. A notable exception is the JVM, which forbids most kinds of pointer manipulation. This is done to enhance program security; arbitrary pointer manipulation can lead to unexpected and catastrophic errors.

Conventional computer architectures include some form of “load address” instruction. That is, rather than loading a register with the byte or word at a particular address, the address itself is loaded. Load address instructions are ideal for implementing the `&` (address of) operator. If `V` is a global variable with label `L`, `&V` evaluates to the address `L` represents. The expression `&V` can be implemented as

```
LoadAddress Register, L.
```

If `W` is a local variable at offset `F` in the current frame, then `&W` represents `F` plus the current value of the frame pointer. Thus `&W` can be implemented as

```
LoadAddress Register, F+FramePointer.
```

The dereference operator, unary `*`, takes a pointer and fetches the value at the address the pointer contains. Thus `*P` is evaluated in two steps. First `P` is evaluated, loading an address (`P`'s value) into a register or temporary. Then that

address is used to fetch a data value. The following instruction pair is commonly generated to evaluate $*P$

```
Load Reg1, P      ; Put P's value into Reg1
Load Reg2, (Reg1) ; Use Reg1's contents as an address
                  ; to load Register 2
```

Pointer arithmetic is also allowed in C and C++. If P is a pointer, the expressions $P + i$ and $P - i$ are allowed, where i is an integer value. In these cases, P is treated as an unsigned integer. The value of i is multiplied by the `sizeof(T)` where P 's type is $*T$. Then $i * \text{sizeof}(T)$ is added to or subtracted from P . For example, if P is of type `*int` (a pointer to `int`) and `ints` require 4 bytes, then the expression $P+1$ generates

```
Load Reg1, P      ; Put P's value into Reg1
AddU Reg2, Reg1,4 ; Unsigned add of 4 (sizeof(int)) to P
```

Since pointer arithmetic is limited to these two special cases, it is easiest implemented by extending the code generation subroutines for addition and subtraction to allow pointer types (as well as integer and floating types).

C++ and Java provide references which are a restricted form of pointer. In C++ a reference type (`T&`), once initialized, cannot be changed. In Java, a reference type may be assigned to, but only with valid references (pointers) to heap-allocated objects.

In terms of code generation, references are treated almost identically to explicit pointers. The only difference is that no explicit dereference operators appear in the AST. That is, if P is of type `int*` and R is of type `int&`, the two

expressions $(*P)+1$ and $R+1$ do exactly the same thing. In fact, it is often a good translation strategy to include an explicit “dereference” AST node in an AST that contains references. These nodes, added during type checking, can then cue the code generator that the value of a reference must be used as a pointer to fetch a data value from memory.

Some programming languages, like Pascal and Modula, provide for “var” or “reference” parameters. These parameters are implemented just as reference values are; they represent an address (of an actual parameter) which is used to indirectly fetch a data value when a corresponding formal parameter is used.

12.8 Type Conversion

During translation of an expression, it may be necessary to convert the type of a value. This may be forced by a type cast (e.g., `(float) 10`) or by the type rules of the language (`1 + 1.0`). In any event, we’ll assume that during translation and type checking, explicit type conversion AST nodes have been inserted where necessary to signal type conversions. The `ConvertNode` AST node (see Figure 12.13) signals that the `Operand` node (an expression) is to be converted to the type specified by the `Type` field after evaluation. Type-checking has verified that the type conversion is legal.

Some type conversion require no extra code generation. For example, on the JVM (and most other machines) bytes, half words and full words are all manipulated as 32 bit integers on the stack or within registers. Thus the two expressions `'A'+1` and `(int) 'A'+1` generate the same code.

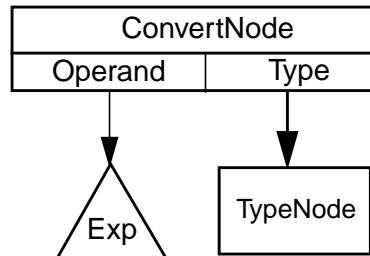


Figure 12.13 Abstract Syntax Tree for Type Conversion

Conversions between single and double length operands and integer and floating representations *do* require additional code. On the JVM special instructions are provided; on other machines library subroutines may need to be called. Four distinct internal representations must be handled—32 bit integer, 64 bit integer, 32 bit floating point and 64 bit floating point. Conversions between each of these representations must be supported.

On the JVM 12 conversion instructions of the form $x2y$ are provided, where x and y can be i (for integer), l (for long integer), f (for float) and d (for double). Thus $i2f$ takes an integer at the top of the stack and converts it to a floating point value.

Three additional conversion instructions, $i2s$, $i2b$ and $i2c$ handle explicit casts of integer values to short, byte and character forms.

12.9 Assignment Operators

In Java, C and C++ the assignment operator, $=$, may be used anywhere within an expression; it may also be cascaded. Thus both $a+(b=c)$ and $a=b=c$ are legal.

Java also provides an assignment statement. In contrast, C and C++ allow an expression to be used wherever a statement is expected. At the statement level, an assignment discards its right-hand side value after the left-hand side variable is updated.

The AST for an assignment to a simple scalar variable is shown in Figure 12.14. Code generation is reasonably straightforward. We do need to be careful about how we handle the variable that is the target of the assignment.

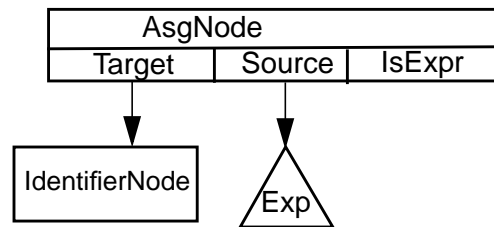


Figure 12.14 Abstract Syntax Tree for Assignments

Normally, we recursively apply the code generator to all the children of an AST node. Were we to do this the `IdentifierNode`, we'd compute its value and put in on the stack or in a register. This *isn't* what we want. For simple variables, we'll just extract the identifier's label or frame offset directly from the `IdentifierNode`. In more complex cases (e.g., when the left-hand side is a subscripted variable or pointer expression), we need a special variant of our code generator that evaluates the *address* denoted by an expression rather than the *value* denoted by an expression. To see why this distinction is necessary, observe that `a[i+j]` on the left-hand side of an assignment represents the *address* of an array element; on the right-hand side it represents the *value* of an array element. Whenever we trans-

late an expression or construct, we'll need to be clear about whether we want to generate code to compute an address or a value.

We'll also need to be sure that the expression value to be assigned in on the stack or in a register. Normally, translating an expression does just that. Recall however, that as an optimization, we sometimes delay loading a literal value or translate a boolean-valued expression in jump code form. We'll need to be sure we have a value to store before we generate code to update the left-hand side's value.

Since we are handling only simple variables, translating an assignment is simple. We just evaluate (as usual) the right-hand side of the expression and store its value in the variable on the left-hand side. The value of the expression is the value of the right-hand side. The `IsExpr` field (a boolean) marks whether the assignment is used as an expression or a statement. If the assignment is used as a statement, the right-hand side value, after assignment, may be discarded. If the assignment is used as an expression, the right-hand side value will be preserved.

We will use two code generation subroutines:

```
GenStoreGlobal(Source, Type, Label, PreserveSource),
```

```
GenStoreLocal(Source, Type, Offset, PreserveSource).
```

`GenStoreGlobal` will generate a store of `Source` into a global variable using the label and type specified. `GenStoreLocal` will generate a store of `Source` into a local variable using the offset and type specified.

In both routines, `Source` is an `AddressNode`. It may represent a temporary or literal. In the case of a literal, special case instructions may be generated (like a

“store zero” instruction). Otherwise, the literal is loaded into a temporary and stored.

If `PreserveSource` is true, the contents of `Source` is preserved. That is, if `Source` is in a register, that register is not released. If `Source` is on the stack, its value is duplicated so that the value may be reused after the assignment (store instructions automatically pop the stack in the JVM). If `PreserveSource` is false, `Source` is not preserved; if it represents a register, that register is freed after the assignment.

Translation of an assignment operator is detailed in Figure 12.15.

```
AsgNode.CodeGen( )
1. Source.CodeGen()
2. if Source.Result.AccessMode ∈ {JumpIfTrue, JumpIfFalse}
3.   then Result ← ConvertFromJumpCode(Result,
                                     Source.Result.AccessMode, Source.Result.Label)
4.   else Result ← Source.Result
5. if Target.Address.AccessMode = Global
6.   then GenStoreGlobal(Result, Type, Target.Address.Label, IsExpr)
7.   else GenStoreLocal(Result, Type, Target.Address.Offset, IsExpr)
```

Figure 12.15 Code Generation for Assignments

As an example, consider $A = B = C + 1$ where A , B and C are local integers with frame indices of 1, 2 and 3 respectively. First, $C + 1$ is evaluated. Since $B = C + 1$ is used as an expression, the value of $C + 1$ is duplicated (and thereby preserved) so that it can also be assigned to A . The JVM code we generate is

```
iload      3 ; Push local #3 (C) onto the stack
iconst_1   ; Push 1 onto the stack
iadd      ; Compute C + 1
dup       ; Duplicate C + 1 for second store
```

```
istore    2 ; Store top of stack into local #2 (B)
istore    1 ; Store top of stack into local #1 (A)
```

In C and C++ we have a problem whenever an expression is used where a statement is expected. The difficulty is that a value has been computed (into a register or onto the stack), but that value will never be used. If we just ignore the value a register will never be freed (and hence will be “lost” for the rest of the program) or a stack value will never be popped.

To handle this problem, we’ll assume that whenever an expression is used as a statement, a `VoidExpr` AST node will be placed above the expression. This will signal that the expression’s value won’t be used, and the register or stack location holding it can be released. The `VoidExpr` node can also be used for the left operand of the “,” (sequencing) operator, where the left operand is evaluated for side-effects and then discarded.

Note that in some cases an expression used as a statement need not be evaluated at all. In particular, if the expression has no side-effects, there is no point to evaluating an expression whose value will be ignored. If we know that the expression does no assignments, performs no input/output and makes no calls (which themselves may have side-effects), we can delete the AST as useless. This analysis might be done as part of a pre-code generation simplification phase or as a data flow analysis that detects whether or not a value is dead (see Chapter 16).

12.9.1 Compound Assignment Operators

Java, C and C++ contain a useful variant of the assignment operator—the compound assignment operator. This operator uses its left operand twice—first to obtain an operand value, then as the target of an assignment. Informally $a \text{ op} = b$ is equivalent to $a = a \text{ op } b$ except that a must be evaluated only once. For simple variables the left-hand side may be visited twice, first to compute a value and then to obtain the address used in a store instruction. If evaluation of the left-hand side's address is non-trivial, we first evaluate the variable's address. We then use that address twice, as the source of an operand and as the target of the result.

The code generator for $+=$ is shown in Figure 12.16. Other compound assignment operators are translated in a similar manner, using the operator/op code mapping shown in Figure 12.7.

```

PlusAsgNode.CodeGen( )
1. LeftOperand.CodeGen()
2. RightOperand.CodeGen()
3. Result ← GenAdd(Result, Type,
                   LeftOperand.Result, RightOperand.Result)
4. FreeTemp(LeftOperand.Result, RightOperand.Result)
5. if LeftOperand.Address.AccessMode = Global
6.   then GenStoreGlobal(Result, Type,
                        LeftOperand.Address.Label, IsExpr)
7.   else GenStoreLocal(Result, Type,
                       LeftOperand.Address.Offset, IsExpr)

```

Figure 12.16 Code Generation for $+=$ Operator

As an example, consider $A += B += C + 1$ where A , B and C are local integers with frame indices of 1, 2 and 3 respectively. The JVM code we generate is

```

iload    1  ; Push local #1 (A) onto the stack
iload    2  ; Push local #2 (B) onto the stack
iload    3  ; Push local #3 (C) onto the stack

```

```
iconst_1    ; Push 1 onto the stack
iadd        ; Compute C+1
iadd        ; Compute B+C+1
dup         ; Duplicate B+C+1
istore 2    ; Store top of stack into local #2 (B)
iadd        ; Compute A+B+C+1
istore 1    ; Store top of stack into local #1 (A)
```

12.9.2 Increment and Decrement Operators

The increment (`++`) and decrement (`--`) operators are widely used in Java, C and C++. The prefix form `++a` is identical to `a += 1` and `--a` is identical to `a -= 1`. Thus these two operators may be translated using the techniques of the previous section.

The postfix forms, `a++` and `a--` are a bit different in that after incrementing or decrementing the variable, the *original* value of the variable is returned as the value of the expression. This difference is easy to accommodate during translation.

If `a++` is used as a statement (as it often is), a translation of `a += 1` may be used (this holds for `a--` too).

Otherwise, when `a` is evaluated, its value is duplicated on the stack, or the register holding its value is retained. The value of `a` is then incremented or decre-

mented and stored back into `a`. Finally, the original value of `a`, still on the stack or in a register, is used as the value of the expression.

For example, consider `a+(b++)`, where `a` and `b` are integer locals with frame indices of 1 and 2. The following JVM code would be generated

```
iload      1 ; Push local #1 (A) onto the stack
iload      2 ; Push local #2 (B) onto the stack
dup                ; Duplicate B
iconst_1         ; Push 1 onto the stack
iadd            ; Compute B+1
istore      2 ; Store B+1 into local #2 (B)
iadd            ; Compute A+original value of B
```

As a final observation, note that the definition of the postfix `++` operator makes `C++`'s name something of a misnomer. After improving `C`'s definition we certainly don't want to retain the *original* language!

Exercises

1. Most C and Java compilers provide an option to display the assembly instructions that are generated. Compile the following procedure on your favorite C or Java compiler and get a listing of the code generated for it. Examine each instruction and explain what step it plays in implementing the procedure's body.

```
void proc() {  
    int a=1,b=2,c=3,d=4;  
    a = b + c * d;  
    c=(a<b)?1:2;  
    a=b++ + ++c;  
}
```

2. Show the code that would be generated for each of the following expressions. Assume I and J are integer locals, K and L are integer globals, A and B are floating point locals and C and D are floating point globals.

$I+J+1$

$K-L-1$

$A+B+1$

$C-D-1$

$I+L+10$

$A-D-10$

$I+A+5$

$I+(1+2)$

$I+1+2$

3. The code generated for $I+1+2$ in Exercise 2 is probably not the same as that generated for $I+(1+2)$. Why is this? What must be done so that both generate the same (shorter) instruction sequence?
4. Some languages, including Java, require that all variables be initialized. Explain how to extend the code generator for `VarDeclNodes` (Figure 12.3) so that both local and global variables are given a default initialization if no explicit initial value is declared.
5. Complete the code generator for `VarDeclNodes` (Figure 12.3) by inserting calls to the proper code generation subroutines to store the value of `InitValue` into a local variable.
6. On register-oriented machines, `GetTemp` is expected to return a different register each time it is called. This means that `GetTemp` may run out of registers if too many registers are requested before `FreeTemp` is called.

What can be done (other than terminating compilation) if `GetTemp` is called when no more free registers are available?

7. It is clear that some expressions are more complex than others in terms of the number of registers or stack locations they will require.

Explain how an AST representing an expression can be traversed to determine the number of registers or stack locations its translation will require. Illustrate your technique on the following expression

$((A+B) + (C+D)) + ((E+F) + (G+H))$

8. Explain how to extend the code generator for the `++` and `--` operators to handle pointers as well as numeric values.
9. Assume we add a new operator, the `swap` operator, `<->`, to Java or C. Given two variables, `v1` and `v2`, the expression `v1<->v2` assigns `v2`'s value to `v1` and assigns `v1`'s original value to `v2`, and then returns `v1`'s new value.

Define a code generator that correctly implements the `swap` operator.

10. Show the jump code that would be generated for each of the following expressions. Assume `A`, `B` and `C` are all integer locals, and that `&&` and `||` are left-associative.

$(A < 1) \&\& (B > 1) \&\& (C \neq 0)$

$(A < 1) || (B > 1) || (C \neq 0)$

$(A < 1) \&\& (B > 1) || (C \neq 0)$

$(A < 1) || (B > 1) \&\& (C \neq 0)$

11. What is generated if the boolean literals `true` and `false` are translated into `JumpIfTrue` or `JumpIfFalse` form?

What code is generated for the expressions `(true?1:2)` and `(false?1:2)`? Can the generated code be improved?

12. If we translate the expression `(double) (long) i` (where `i` is an `int`) for the JVM we will generate an `i2l` instruction (`int` to `long`) followed by an `l2d` (`long` to `double`) instruction. Can these two instructions be combined into an `i2d` (`int` to `double`) instruction?

Now consider `(double) (float) i`. We now will generate an `i2f` instruction (`int` to `float`) followed by an `f2d` (`float` to `double`) instruction. Can these two instructions be combined into an `i2d` instruction?

In general what restrictions must be satisfied to allow two consecutive type conversion instructions to be combined into a single type conversion instruction?

13. In C and C++ the expression `exp1, exp2` means evaluate `exp1`, then `exp2` and return the value of `exp2`. If `exp1` has no side effects (assignments, I/O or system calls) it need not be evaluated at all. How can we test `exp1`'s AST, prior to code generation, to see if we can suppress its evaluation?
14. On some machines selected registers are preassigned special values (like 0 or 1) for the life of a program. How can a code generator for such a machine exploit these preloaded registers to improve code quality?
15. It is sometimes the case that a particular expression value is needed more than once. An example of this is the statement `a[i+1]=b[i+1]+c[i+1]` where `i+1` is used three times.

An expression that may be reused is called a **redundant** or **common** expression. An AST may be transformed so that an expression that is used more than once has multiple parents (one parent for each context that requires the same value). An AST node that has more than one parent isn't really a tree anymore. Rather, it's a **directed-acyclic graph** (a **dag**).

How must code generators for expressions be changed when they are translating an AST node that has more than one parent?

13

Code Generation: Control Structures and Subroutines

13.1 Code Generation for Control Structures

Control structures, whether conditionals like if and switch statements, or while and for loops, involve generation of conditional branch instructions. That is, based on a boolean value or relation, statements may be executed or not. In translating these constructs we will utilize jump code just as we did for conditional expressions. Depending on the nature of the construct, we will utilize either `JumpIfTrue` or `JumpIfFalse` form.

13.1.1 If Statements

The AST corresponding to an if statement is shown in Figure 13.1. An `IfNode` has three subtrees, corresponding to the condition controlling the if, the then statements and the else statements. The code we will generate will be of the form

1. If `condition` is false go to 4.
2. Execute statements in `thenPart`.
3. Go to 5.
4. Execute statements in `elsePart`.
5. Statements following the if statement.

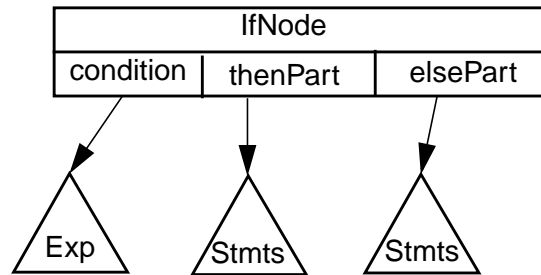


Figure 13.1 Abstract Syntax Tree for an if statement

At run-time the code we generate first evaluates the boolean-valued condition. If it is false, the code branches to a label at the head of the instructions generated for the else part. If the condition is true, no branch is taken; we “fall through” to the instructions generated for the then statements. After this code is executed, we branch to the very end of the if statement (to avoid incorrectly executing the statements of the else part).

The `CodeGen` function for an `IfNode` is shown in Figure 13.2.

As an example, consider the following statement

```
if (b) a=1; else a=2;
```

Assume `a` and `b` are local variables with indices of 1 and 2. We know `b` is a boolean, limited to either 1 (true) or 0 (false). The code we generate is:

IfNode.CodeGen()

1. ElseLabel ← CreateUniqueLabel()
2. condition.Result ← AddressNode(JumpIfFalse, ElseLabel)
3. condition.CodeGen()
4. thenPart.CodeGen()
5. OutLabel ← CreateUniqueLabel()
6. GenGoTo(OutLabel)
7. GenLabel(ElseLabel)
8. elsePart.CodeGen()
9. GenLabel(OutLabel)

Figure 13.2 Code Generation for If Statements Using Jump Code

```

iload      2 ; Push local #2 (b) onto the stack

ifeq      L1 ; Goto L1 if b is 0 (false)

iconst_1   ; Push 1

istore     1 ; Store stack top into local #1 (a)

goto      L2 ; Skip around else statements

L1: iconst_2   ; Push 2

        istore     1 ; Store stack top into local #1 (a)

L2:

```

Improving If Then Statements The code we generate for `IfNodes` is correct even if the `elsePart` is a `nullNode`. This means we correctly translate both “if then” and “if then else” statements. However, the code for an if then statement is a bit clumsy, as we generate a `goto` to the immediately following statement. Thus for

```
if (b) a=1;
```

we generate

```

iload      2 ; Push local #2 (B) onto the stack

ifeq      L1 ; Goto L1 if B is 0 (false)

```

```

        iconst_1      ; Push 1
        istore       1 ; Store stack top into local #1 (a)
        goto        L2 ; Skip around else statements

L1:
L2:

```

We can improve code quality by checking whether the `elsePart` is a `nullNode` before generating a `goto` to the end label, as shown in Figure 13.3.

```

IfNode.CodeGen( )
1. ElseLabel ← CreateUniqueLabel()
2. condition.Result ← AddressNode(JumpIfFalse, ElseLabel)
3. condition.CodeGen()
4. thenPart.CodeGen()
5. if elsePart = null
6.     then GenLabel(ElseLabel)
7.     else OutLabel ← CreateUniqueLabel()
8.         GenGoTo(OutLabel)
9.         GenLabel(ElseLabel)
10.    elsePart.CodeGen()
11.    GenLabel(OutLabel)

```

Figure 13.3 Improved Code Generation for If Statements

13.1.2 While, Do and Repeat Loops

The AST corresponding to a while statement is shown in Figure 13.4. A `while-Node` has two subtrees, corresponding to the condition controlling the loop and the loop body. A straightforward translation is

1. If `condition` is false go to 4.
2. Execute statements in `loopBody`.
3. Go to 1.
4. Statements following the while loop.

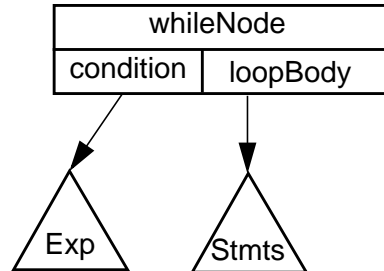


Figure 13.4 Abstract Syntax Tree for a While Statement

This translation is correct, but it is not as efficient as we would like. We expect a loop to iterate many times. Each time through the loop we will execute an unconditional branch to the top of the loop to reevaluate the loop control expression, followed by a conditional branch that will probably fail (it succeeds only after the last iteration). A more efficient alternative is to generate the conditional expression and conditional branch *after* the loop body, with an initial goto around the loop body (so that zero iterations of the loop are possible). That is, we generate code structured as

1. Go to 3.
2. Execute statements in loopBody.
3. If condition is true go to 2.

Now our translation executes only one branch per iteration rather than two.

While loops may contain a continue statement, which forces the loop's termination condition to be evaluated and tested. As detailed in Section 13.1.4, we assume that the function `getContinueLabel()` will return the label used to mark the target of a continue.

The code generator for while loops is shown in Figure 13.5.

```

WhileNode.CodeGen( )
1.  ConditionLabel ← getContinueLabel()
2.  GenGoTo(ConditionLabel)
3.  TopLabel ← CreateUniqueLabel()
4.  GenLabel(TopLabel)
5.  loopBody.CodeGen()
6.  GenLabel(ConditionLabel)
7.  condition.Result ← AddressNode(JumpIfTrue, TopLabel)
8.  condition.CodeGen()

```

Figure 13.5 Code Generation for While Statements

As an example, consider the following while loop, where *I* is an integer local with a variable index of 2 and *L1* is chosen as the *ConditionLabel*

```
while (I >= 0) { I--; }
```

The JVM code generated is

```

goto          L1 ; Skip around loop body

L2:
  iload       2 ; Push local #2 (I) onto the stack
  iconst_1    ; Push 1
  isub        ; Compute I-1
  istore      2 ; Store I-1 into local #2 (I)

L1:
  iload       2 ; Push local #2 (I) onto the stack
  ifge        L2 ; Goto L2 if I is >= 0

```

Do and Repeat LoopsJava, C and C++ contain a variant of the while loop—the do while loop. A do while loop is just a while loop that evaluates and tests its termi-

nation condition after executing the loop body rather than before. In our translation for while loops we placed evaluation and testing of the termination condition after the loop body—just where the do while loop wants it!

We need change very little of the code generator we used for while loops to handle do while loops. In fact, all we need to do is to eliminate the initial goto we used to jump around the loop body. With that change, the remaining code will correctly translate do while loops. For example, the statement

```
do { I--;} while (I >= 0);
```

generates the following (label L1 is generated in case the loop body contains a continue statement):

```
L2:
    iload      2 ; Push local #2 (I) onto the stack
    iconst_1   ; Push 1
    isub      ; Compute I-1
    istore     2 ; Store I-1 into local #2 (I)

L1:
    iload      2 ; Push local #2 (I) onto the stack
    ifge      L2 ; Goto L2 if I is >= 0
```

Some languages, like Pascal and Modula 3, contain a repeat until loop. This is essentially a do while loop except for the fact that the loop is terminated when the control condition becomes false rather than true. Again, only minor changes are needed to handle a repeat loop. As was the case for do loops, the initial branch

around the loop body is removed. This is because repeat loops always test for termination at the end of an iteration rather than at the start of an iteration.

The only other change is that we wish to continue execution if the condition is false, so we evaluate the termination condition in the `JumpIfFalse` form of jump code. This allows the repeat until loop to terminate (by falling through) when the condition becomes true.

13.1.3 For Loops

For loops are translated much like while loops. As shown in Figure 13.6, the AST for a for loop adds subtrees corresponding to loop initialization and increment.

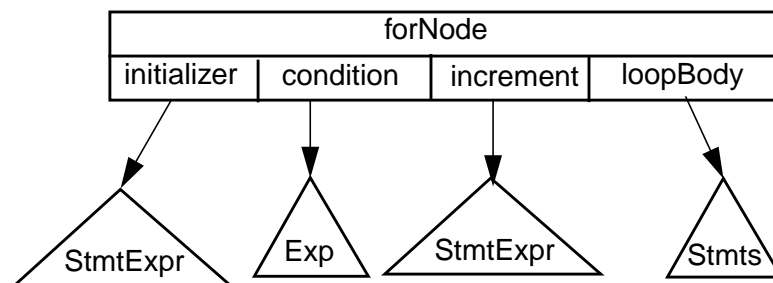


Figure 13.6 Abstract Syntax Tree for a For Statement

For loops are expected to iterate many times. Therefore after executing the loop initializer, we again skip past the loop body and increment code to reach the termination condition, which is placed at the bottom of the loop:

1. Execute `initializer` code.
2. Go to 5.
3. Execute statements in `loopBody`.
4. Execute `increment` code.
5. If `condition` is true go to 3.

Any or all of the expressions used to define loop initialization, loop increment or loop terminations may be null. Null initialization or increment expressions are no problem. They appear as `nullNodes` in the AST and generate nothing when `CodeGen` is called. However, the termination condition is expected to produce a conditional branch to the head of the loop. If it is null, we must generate an unconditional branch back to the top of the loop body. (For loops without a termination condition can only be exited with a `break` or `return` statement within the loop body.)

The code generator shown in Figure 13.7 is used to translate a `forNode`.

As an example, consider the following for loop (where `i` and `j` are locals with variable indices of 1 and 2)

```

ForNode.CodeGen( )
1.  initializer.CodeGen()
2.  SkipLabel ← CreateUniqueLabel()
3.  GenGoTo(SkipLabel)
4.  TopLabel ← CreateUniqueLabel()
5.  GenLabel(TopLabel)
6.  loopBody.CodeGen()
7.  ContinueLabel ← getContinueLabel()
8.  GenLabel(ContinueLabel)
9.  increment.CodeGen()
10. GenLabel(SkipLabel)
11. if condition = null
12.   then GenGoTo(TopLabel)
13.   else condition.Result ← AddressNode(JumpIfTrue, TopLabel)
14.       condition.CodeGen()

```

Figure 13.7 Code Generation for For Loops

```

for (i=100;i!=0;i--) {
    j = i;
}

```

The JVM code we generate is

```

bipush      100; Push 100 onto the stack

istore     1  ; Store 100 into local #1 (i)

goto      L1 ; skip around loop body and increment

L2:

iload     1  ; Push local #1 (i) onto the stack

istore     2  ; Store i into local #2 (j)

L3:                ; Target label for continue statements

iload     1  ; Push local #1 (i) onto the stack

iconst_1   ; Push 1

isub      ; Compute i-1

```

```

    istore    1 ; Store i-1 into local #1 (i)
L1:
    iload    1 ; Push local #1 (i) onto the stack
    ifne    L2 ; Goto L2 if i is != 0

```

The special case of

```
for (;;) { }
```

(which represents an infinite loop) is properly handled, generating

```

    goto    L1
L1: L2: L3:
    goto    L2

```

Java and C++ allow a local declaration of a loop index as part of initialization, as illustrated by the following for loop

```

for (int i=100; i!=0; i--) {
    j = i;
}

```

Local declarations are automatically handled during code generation for the initialization expression. A local variable is declared within the current procedure with a scope limited to the body of the loop. Otherwise translation is identical.

13.1.4 Break, Continue, Return and Goto Statements

Java contains no `goto` statement. It does, however, include `break` and `continue` statements which are restricted forms of a `goto`, as well as a `return` statement. We'll consider the `continue` statement first.

Continue Statements Like the `continue` statement found in C and C++, Java's `continue` statement attempts to "continue with" the next iteration of a `while`, `do` or `for` loop. That is, it transfers control to the bottom of a loop where the loop index is iterated (in a `for` loop) and the termination condition is evaluated and tested.

A `continue` may only appear within a loop; this is enforced by the parser or semantic analyzer. Unlike C and C++ a loop label may be specified in a `continue` statement. An unlabeled `continue` references the innermost `for`, `while` or `do` loop in which it is contained. A labeled `continue` references the enclosing loop that has the corresponding label. Again, semantic analysis verifies that an enclosing loop with the proper label exists.

Any statement in Java may be labeled. We'll assume an AST node `labeledStmt` that contains a string-valued field `stmtLabel`. If the statement is labeled, `stmtLabel` contains the label in string form. If the statement is unlabeled, `stmtLabel` is null. `labeledStmt` also contains a field `stmt` that is the AST node representing the labeled statement.

```
labeledStmtCodeGen( )
1.  if stmt ∈ { whileNode, doNode, forNode }
2.      then continueList ← continueItem(stmtLabel, CreateUniqueLabel(),
                                     length(finalList), continueList)
3.  stmt.CodeGen()
```

Figure 13.8 Code Generation for Labeled Statements

The code generator of `labeledStmt`, defined in Figure 13.8, checks `stmt` to see if it represents a looping statement (of any sort). If it does, the code generator creates a unique label to be used within the loop as the target of continues. It adds this label, called `asmLabel`, along with the user-defined label (field `stmtLabel`), onto a list called `continueList`. This list also includes an integer `finalDepth` that represents the number of try statements with finally blocks this statement is nested in (see Section 13.2.6) and a field `next` that links to the next element of the list. `continueList` will be examined when a continue statement is translated.

As mentioned earlier, looping statements, when they are translated, call `getContinueLabel()` to get the label assigned for use by continue statements. This label is simply the assembly-level label (generated by `CreateUniqueLabel`) at the head of `continueList`.

We'll also use two new code generation subroutines. `GenJumpSubr(Label)` will generate a subroutine call to `Label`. In JVM this is a `jsr` instruction. `GenFinalCalls(N)` will generate `N` subroutine calls (using `GenJumpSubr`) to the labels contained in the first `N` elements of `finalList`. This list is created by try statements with finally blocks to record the blocks that must be executed prior to executing a continue, break or return statement (see Section Section 13.2.6 for more details).

A continue statement that references no label will use the first element of `continueList`. If the continue is labeled, it will search the `continueList` for a matching label. The field `label` in `continueNode` will contain the label parameter, if any, referenced in the continue. A code generator for continue statements is shown in Figure 13.9.

```

ContinueNode.CodeGen ( )
1.  if stmtLabel = null
2.      then GenFinalCalls(length(finalList) - continueList.finalDepth)
3.          GenGoTo(continueList.asmLabel)
4.      else listPos ← continueList
5.          while listPos.stmtLabel ≠ stmtLabel
6.              do listPos ← listPos.next
7.          GenFinalCalls(length(finalList) - listPos.finalDepth)
8.          GenGoTo(listPos.asmLabel)

```

Figure 13.9 Code Generation for Continue Statements

In most cases continue statements are very easy to translate. Try statements with finally blocks are rarely used, so a continue generates a branch to the “loop continuation test” of the innermost enclosing looping statement or the looping statement with the selected label. In C and C++ things are simpler still—continues don’t use labels so the innermost looping statement is always selected.

Break Statements In Java an unlabeled break statement has the same meaning as the break statement found in C and C++. The innermost while, do, for or switch statement is exited, and execution continues with the statement immediately following the exited statement.

A labeled break exits the enclosing statement with a matching label, and continues execution with that statement’s successor. For both labeled and unlabeled breaks, semantic analysis has verified that a suitable target statement for the break does in fact exist.

We’ll extend the code generator for the `labeledStmt` node to prepare for a break when it sees a labeled statement or an unlabeled while, do, for or switch statement. It will create an assembler label to be used as the target of break state-

ments within the statement it labels. It will add this label, along with the Java label (if any) of the statement it represents to `breakList`, a list representing potential targets of break statements. It will also add a flag indicating whether the statement it represents is a while, do, for or switch statement (and hence is a valid target for an unlabeled break). Also included is an integer `finalDepth` that represents the number of try statements with finally blocks this statement is nested in (see Section 13.2.6).

The code generator will also generate a label definition for the break target after it has translated its labeled statement. The revised code generator for `labeledStmt` is shown in Figure 13.10.

```

labeledStmtCodeGen( )
1.  if stmt ∈ { whileNode, doNode, forNode }
2.      then continueList ← continueItem(stmtLabel, CreateUniqueLabel(),
                                     length(finalList), continueList)
3.  unlabeledBreakTarget ←
       stmt ∈ { whileNode, doNode, forNode, switchNode}
4.  if unlabeledBreakTarget or stmtLabel ≠ null
5.      then breakLabel ← CreateUniqueLabel()
6.      breakList ← breakItem(stmtLabel, breakLabel,
                             unlabeledBreakTarget, length(finalList), breakList)
7.  stmt.CodeGen()
8.  if unlabeledBreakTarget or stmtLabel ≠ null
9.      then GenLabel(breakLabel)

```

Figure 13.10 Revised Code Generator for Labeled Statements

A break statement, if unlabeled, will use the first object on `breakList` for which `unlabeledBreakTarget` is true. If the break is labeled, it will search the `breakList` for a matching label. Field `label` in `breakNode` will contain the label parameter, if any, used with the break. The code generator for break statements is shown in Figure 13.11.

```

BreakNode.CodeGen ( )
1. listPos ← breakList
2. if stmtLabel = null
3.     then while not listPos.unlabeledBreakTarget
4.         do listPos ← listPos.next
5.         GenFinalCalls(length(finalList) - listPos.finalDepth)
6.         GenGoTo(listPos.breakLabel)
7.     else while listPos.stmtLabel ≠ stmtLabel
8.         do listPos ← listPos.next
9.         GenFinalCalls(length(finalList) - listPos.finalDepth)
10.        GenGoTo(listPos.breakLabel)

```

Figure 13.11 Code Generation for Break Statements

In practice break statements are very easy to translate. Try statements with finally blocks are rarely used, so a break normally generates just a branch to the successor of the statement selected by the break. In C and C++ things are simpler still—breaks don't use labels so the innermost switch or looping statement is always selected.

As an example of how break and continue statements are translated, consider the following while loop

```

while (b1) {
    if (b2) continue;
    if (b3) break;
}

```

The following JVM code is generated

```

goto        L1 ; Skip around loop body

L2:
iload      2 ; Push local #2 (b2) onto the stack
ifeq      L3 ; Goto L3 if b2 is false (0)

```

```
goto      L1 ; Goto L1 to continue the loop
L3:
  iload   3  ; Push local #3 (b3) onto the stack
  ifeq    L4 ; Goto L4 if b3 is false (0)
  goto    L5 ; Goto L5 to break out of the loop
L1: L4:
  iload   1  ; Push local #1 (b1) onto the stack
  ifne    L2 ; Goto L2 if b1 is true (1)
L5:
```

Label L1, the target of the continue statement, is created by the `labeledStmt` node that contains the while loop. L1 marks the location where the loop termination condition is tested; it is defined when the while loop is translated. Label L5, the target of the exit statement, is created and defined by the `labeledStmt` node just beyond the end of the while loop. Label L2, the top of the while loop, is created and defined when the loop is translated. Labels L3 and L4 are created and defined by the if statements within the loop's body .

Return StatementsA `returnNode` represents a return statement. The field `returnVal` is null if no value is returned; otherwise it is an AST node representing an expression to be evaluated and returned. If a return is nested within one or more try statements that have finally blocks, these blocks must be executed prior to doing the return. The code generator for a `returnNode` is shown in Figure 13.12. The

code generation subroutine `GenReturn(Value)` generates a return instruction based on `Value`'s type (an `ireturn`, `lreturn`, `freturn`, `dreturn`, or `areturn`). If `Value` is null, an ordinary return instruction is generated.

```
return.CodeGen( )
1.  if returnVal ≠ null
2.      then returnVal.CodeGen()
3.  GenFinalCalls(length(finalList))
4.  GenReturn(returnVal)
```

Figure 13.12 Code Generator for Return Statements

`Goto StatementsJava` contains no `goto` statement, but many other languages, including `C` and `C++`, do. Most languages that allow `gotos` restrict them to be **intraprocedural**. That is, a label and all `gotos` that reference it must be in the same procedure or function. Languages usually do not require that identifiers used as statement labels be distinct from identifiers used for other purposes. Thus in `C` and `C++`, the statement

```
a: a+1;
```

is legal. Labels are usually kept in a separate symbol table, distinct from the main symbol table used to store ordinary declarations.

Labels need not be defined before they are used (so that “forward `gotos`” are possible). Type checking guarantees that all labels used in `gotos` are in fact defined somewhere in the current procedure. To translate a `goto` statement (e.g., in `C` or `C++`), we simply assign an assembler label to each source-level label, the first time that label is encountered. That assembler label is generated when its corresponding source-level label is encountered during code generation. Each source-level

`goto L` is translated to an assembly language `goto` using the assembler label assigned to `L`.

A few languages, like Pascal, allow **non-local gotos**. A non-local goto transfers control to a label outside the current procedure, exiting the current procedure. Non-local gotos generate far more than a single goto instruction due to the overhead of returning from a procedure to its caller.

Since we have not yet studied procedure call and return mechanisms (see Section 13.2.1), we will not detail the exact steps needed to perform a non-local goto. However, a non-local goto can be viewed as a very limited form of exception handling (see Section 13.2.6). That is, a non-local goto in effect throws an exception that is caught and handled within the calling procedure that contains the target label. Hence the mechanisms we develop for exception handling will be suitable for non-local gotos too.

13.1.5 Switch and Case Statements

Java, C and C++ contain a switch statement that allows the selection of one of a number of statements based on the value of a control expression. Pascal, Ada and Modula 3 contain a case statement that is equivalent. We shall focus on translating switch statements, but our discussion applies equally to case statements.

The AST for a switch statement, rooted at a `switchNode` is shown in Figure 13.13.

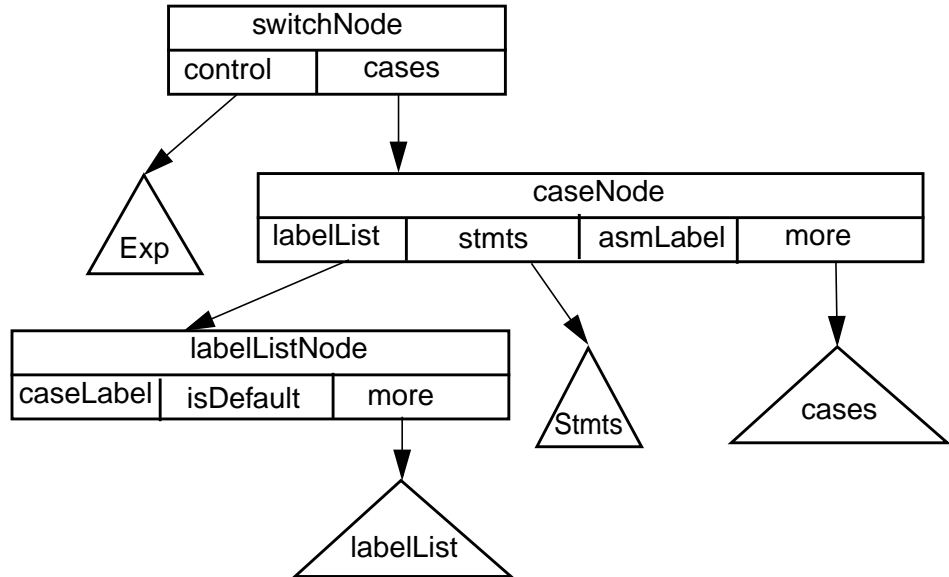


Figure 13.13 Abstract Syntax Tree for a Switch Statement

In the AST `control` represents an integer-valued expression; `cases` is a `caseNode`, representing the cases in the switch. Each `caseNode` has four fields. `labelList` is a `labelListNode` that represents one or more case labels. `stmts` is an AST node representing the statements following a case constant in the switch. `asmLabel` is an assembler label that will be generated with the statements in `stmts`. `more` is either null or another `caseListNode`, representing the remaining cases in the switch.

A `labelListNode` contains an integer field `caseLabel`, a boolean field `isDefault` (representing the default case label) and `more`, a field that is either null or another `labelListNode`, representing the remainder of the list.

Translating a switch involves a number of steps. The control expression has to be translated. Each of the statements in the switch body has to be translated, and assembler labels have to be added at each point marked by a case constant.

Before labeled cases are translated, we'll use the routines shown in Figure 13.14 to build `casePairs`, a list of case constants (field `caseLabel`) and associated assembler labels (field `asmLabel`). That is, if we see `case 123 :` on a statement in the switch, we'll generate an assembler label (say `L234`). Then we'll pair `123` and `L234` together and add the pair to a list of other (`caseLabel`, `asmLabel`) pairs. We'll also assign an assembler label to the default case. If no default is specified, we will create one at the very bottom of the switch body.

The utility routines all serve to “preprocess” a switch statement immediately prior to code generation. `containsDefault(labelList)` examines a list of case labels (a `labelListNode`) to determine if the special default marker is present. `addDefaultIfNecessary(caseNode)` adds a default marker (at the very end of a list of `caseNodes`) if no user-defined default is present. `genCaseLabels(caseNode)` creates a new assembly-level label for each case in a `caseNode`. `getDefaultLabel(caseNode)` extracts the assembly-level label associated with the default case.

`buildCasePairs(labelList, asmLabel)` builds a `casePairs` list for one `labelListNode` given an assembly-level label to be used for all case labels in the node. `buildCasePairs(cases)` takes a `caseNode` and build a `casePairs` list for all the cases the node represents.

The reason we need these utility routines is that we wish to generate code that efficiently selects the assembler label associated with any value computed by the

```

containsDefault( labelList )
1.  if labelList = null
2.      then return false
3.      else return labelList.isDefault or containsDefault(labelList.more)

addDefaultIfNecessary( cases )
1.  if containsDefault(cases.labelList)
2.      then return
3.  elsif cases.more = null
4.      then cases.more ←
           caseNode(labelListNode(0, true, null), null, null, null)
5.      else addDefaultIfNecessary(cases.more)

genCaseLabels( cases )
1.  if cases ≠ null
2.      then cases.asmLabel ← CreateUniqueLabel()
3.      genCaseLabels(cases.more)

getDefaultLabel( cases )
1.  if containsDefault(cases.labelList)
2.      then return cases.asmLabel
3.      else return getDefaultLabel(cases.more)

buildCasePairs( labelList, asmLabel )
1.  if labelList = null
2.      then return null
3.  elsif labelList.isDefault
4.      then return buildCasePairs(labelList.more, asmLabel)
5.      else return casePairs(labelList.caseLabel, asmLabel,
                           buildCasePairs(labelList.more, asmLabel))

buildCasePairs( cases )
1.  if cases = null
2.      then return null
3.      else return appendList( buildCasePairs(cases.labelList, asmLabel),
                           buildCasePairs(cases.more))

```

Figure 13.14 Utility Code Generation Routines for Switch Statements

control expression. For a `switchNode`, `s`, `buildCasePairs(s.cases)` will extract each case constant that appears in the switch and pair it with the assembler label the case constant refers to. We need to decide how to efficiently search this list at

run-time. Since the number and numeric values of case constants can vary greatly, no single search approach is uniformly applicable. In fact, JVM bytecodes directly support two search strategies.

Often the range of case constants that appear in a switch is **dense**. That is, a large fraction of the values in the range from the smallest case constant to the greatest case constant appear as actual case constants. For example, given

```
switch(i) {  
    case 1: a = 1; break;  
    case 3: a = 3; break;  
    case 5: a = 5; break;  
}
```

The case constants used (1, 3, 5) are a reasonably large fraction of the values in the range 1..5. When the range of case constants is dense, a **jump table** translation is appropriate. As part of the translation we build an array of assembler labels (the jump table) indexed by the control expression. The table extends from the smallest case constant used in the switch to the largest. That is, if case constant c labels a statement whose assembly language label is L_i , then $table[c] = L_i$. Positions in the table not explicitly named by a case label get the assembly label of the default case. Control expression values outside the range of the table also select the label of the default case.

As an example, reconsider the above switch. Assume the three cases are assigned the assembly language labels L_1 , L_2 , and L_3 , and the default (a null

statement at the very bottom on the switch) is given label L4. Then the jump table, indexed by values in the range 1..5, contains the labels (L1, L4, L2, L4, L3).

The JVM instruction `tableswitch` conveniently implements a jump table translation of a switch. Its operands are `default`, `low`, `high`, and `table`. `Default` is the label of the default case. `Low` is the lowest case constant represented in `table`. `High` is the highest case constant represented in `table`. `Table` is a table of `high-low+1` labels, each representing the statement to be executed for a particular value in the range `low..high`. `Tableswitch` pops an integer at the top of the JVM stack, and uses it as an index into `table` (if the index is out of range `default` is used). Control is transferred to the code at the selected label.

For the above example we would generate

```
tableswitch default=L4,low=1, high=5,  
           L1, L4, L2, L4, L3
```

Not all switch statements are suitable for a jump table translation. If the value of `high-low` (and hence the size of the jump table) is too large, an unacceptably large amount of space may be consumed by the `tableswitch` instruction.

An alternative is to search a list of case constant, assembler label pairs. If the control expression's value matches a particular case constant, the corresponding label is selected and jumped to. If no match is found, a default label is used. If we sort the list of case constant, assembler label pairs based on case constant values, we need not do a linear search; a faster binary search may be employed.

The JVM instruction `lookupswitch` conveniently implements a search table translation of a switch. Its operands are `default`, `size`, and `table`. `default` is

the label of the default case. `Size` is the number of pairs in `table`. `Table` is a sorted list of case constant, assembler label pairs, one pair for each case constant in the switch. `lookupswitch` pops an integer at the top of the JVM stack, and searches for a match with the first value of each pair in `table` (if no match is found `default` is used). Control is transferred to the code at the label paired with the matching case constant.

For example, consider

```
switch(i) {  
    case -1000000: a = 1; break;  
    case 0:       a = 3; break;  
    case 1000000: a = 5; break;  
}
```

where the three cases are assigned the assembly language labels `L1`, `L2`, and `L3`, and the default (a null statement at the very bottom on the switch) is given label `L4`. We would generate

```
lookupswitch default=L4,size=3,  
    -1000000:L1,  
    0:L2,  
    1000000: L3
```

The decision as to whether to use a jump table or search table translation of a switch statement is not always clear-cut. The most obvious factor is the size of the jump table and the fraction of its entries filled with non-default labels. However, other factors, including the programmer's goals (size versus speed) and imple-

mentation limitations may play a role. We'll assume that a predicate `generateJumpTable(switchNode)` exists that decides whether to use a jump table for a particular switch statement. The details of how `generateJumpTable` makes its choice are left to individual implementors.

No matter which implementation we choose, a variety of utility routines will be needed to create a jump table or search table from a `casePairs` list.

`sortCasePairs(casePairsList)` will sort a `casePairs` list into ascending order, based on the value of `caseLabel`. `getMinCaseLabel(casePairsList)` will extract the minimum case label from a sorted `casePairs` list. `getMaxCaseLabel(casePairsList)` will extract the maximum case label from a sorted `casePairs` list. `genTableswitch(defaultLabel, low, high, table)` generates a `tableswitch` instruction given a default label, a low and high jump table index, and a jump table (encoded as an array of assembly labels). Finally, `genLookupswitch(defaultLabel, size, casePairsList)` generates a `lookupswitch` instruction given a default label, a list size, a list of `casePairs`.

We can now complete the translation of a switch statement, using the code generators defined in Figure 13.15.

As an example of code generation for a complete switch statement consider

```

buildJumpTable( casePairsList, defaultLabel )
1.  min ← getMinCaseLabel(casePairsList)
2.  max ← getMaxCaseLabel(casePairsList)
3.  table ← string[max-min+1]
4.  for   i ← min to max
5.      do if casePairList.caseLabel = i
6.          then table[i-min] ← casePairsList.asmLabel
7.             casePairsList ← casePairsList.next
8.          else table[i-min] ← defaultLabel
8.  return table

caseNode.CodeGen( )
1.  GenLabel(asmLabel)
2.  stmts.CodeGen()
3.  if more ≠ null
4.      then more.CodeGen

switchNode.CodeGen( )
1.  control.CodeGen()
2.  addDefaultIfNecessary (cases)
3.  genCaseLabels (cases)
4.  list ← buildCasePairs(cases)
5.  list ← sortCasePairs(list)
6.  min ← getMinCaseLable(list)
7.  max ← getMaxCaseLable(list)
8.  if generateJumpTable(cases)
9.      then genTableswitch(getDefaultLabel(cases), min, max,
                          buildJumpTable(list, getDefaultLabel(cases)))
10.     else genLookupswitch(getDefaultLabel(cases), length(list), list)
11.  cases.CodeGen()

```

Figure 13.15 Code Generation Routines for Switch Statements

```

switch(i) {
    case 1:  a = 1;  break;
    case 3:  a = 3;  break;
    case 5:  a = 5;  break;
    default: a = 0;
}

```

Assuming that *i* and *a* are locals with variable indices of 3 and 4, the JVM code that we generate is

```

        iload        1  ; Push local #3 (i) onto the stack
        tableswitch default=L4, low=1, high=5,
                    L1, L4, L2, L4, L3
L1:  iconst_1        ; Push 1
        istore       4  ; Store 1 into local #4 (a)
        goto         L5 ; Break
L2:  iconst_3        ; Push 3
        istore       4  ; Store 3 into local #4 (a)
        goto         L5 ; Break
L3:  iconst_5        ; Push 5
        istore       4  ; Store 5 into local #4 (a)
        goto         L5 ; Break
L4:  iconst_0        ; Push 0
        istore       4  ; Store 0 into local #4 (a)
L5:

```


Note that **Java**, like **C** and **C++**, requires the use of a `break` after each case statement to avoid “falling into” remaining case statements. Languages like **Pascal**, **Ada** and **Modula 3** automatically force an exit from a case statement after it is executed; no explicit `break` is needed. This design is a bit less error-prone, though it does preclude the rare circumstance where execution of a whole list of case statements really is wanted. During translation, the code generator for a case statement automatically includes a `goto` after each case statement to a label at the end of the case.

Ada also allows a case statement to be labeled with a range of values (e.g., `1..10` can be used instead of ten distinct constants). This is a handy generalization, but it does require a change in our translation approach. Instead of pairing case constants with assembly labels, we pair **case ranges** with labels. A single case constant, `c`, is treated as the trivial range `c..c`. Now when `casePairs` lists are sorted and traversed, we consider each value in the range instead of a single value.

13.2 Code Generation for Subroutine Calls

Subroutines, whether they are procedures or functions, whether they are recursive or non-recursive, whether they are members of classes or independently declared, form the backbone of program organization. It is essential that we understand how to effectively and efficiently translate subroutine bodies and calls to subroutines.

13.2.1 Parameterless Calls

Since calls in their full generality can be complicated, we'll start with a particularly simple kind of subroutine call—one without parameters. We'll also start with calls to static subroutines; that is, subroutines that are individually declared or are members of a class rather than a class instance.

Given a call `subr()`, what needs to be done? To save space and allow recursion, subroutines are normally translated in **closed** form. That is, the subroutine is translated only once, and the same code is used for all calls to it. This means we must branch to the start of the subroutine's code whenever it is called. However, we must also be able to get back to the point of call after the subroutine has executed. Hence, we must capture a **return address**. This can be done by using a special "subroutine call" instruction that branches to the start address of a subroutine *and* saves the return address of the caller (normally this is the next instruction after the subroutine call instruction).

As we learned in Section 11.2, calls normally push a **frame** onto the run-time stack to accommodate locals variables and control information. Frames are typically accessed via a **frame pointer** that always points to the currently active frame. Hence updating the stack pointer (to push a frame for the subroutine being called) and the frame pointer (to access the newly pushed frame) are essential parts of a subroutine call.

After a subroutine completes execution, the subroutine's frame must be popped from the stack and the caller's stack top and frame pointer must be restored. Then control is returned to the caller, normally to the instruction immediately following the call instruction that began subroutine execution.

If the subroutine called is a function, then a **return value** must also be provided. In **Java** (and many other languages) this is done by leaving the return value on the top of the caller's stack, where operands are normally pushed during execution. On register-oriented architectures, return values are often placed in specially designated registers.

In summary then, the following steps must be done to call a parameterless subroutine.

1. The caller's return address is established and control is passed to the subroutine.
2. A frame for the subroutine is pushed onto the stack and the frame pointer is updated to access the new frame.
3. The caller's stack top, frame pointer, and return address are saved in the newly pushed frame.
4. The subroutine's body is executed.
5. The subroutine's frame is popped from the stack.
6. The caller's stack top, frame pointer and return address are restored.
7. Control is passed to the caller's return address (possibly with a function return value).

The run-time stack prior to step 1 (and after step 7) is illustrated in Figure 13.16(a). The caller's frame is at the top of the stack, accessed by the frame pointer. The run time stack during the subroutine's execution (step 4) is shown in Figure 13.16(b). A frame for the subroutine (the callee) has been pushed onto the

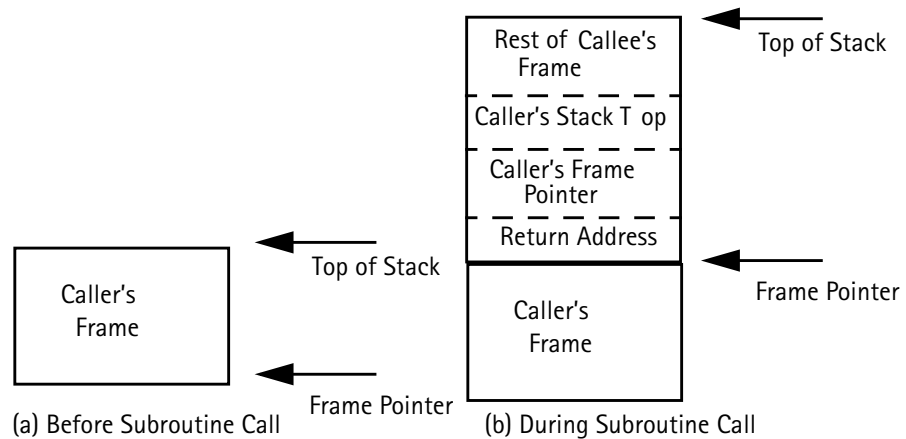


Figure 13.16 Run-time Stack during a Call

stack. It is now accessed through the frame pointer. The caller's stack top, frame pointer and return address are stored within the current frame. The values will be used to reset the stack after the subroutine completes and its frame is popped.

On most computers each of the steps involved in a call takes an instruction or two, making even a very simple subroutine call non-trivial. The JVM is designed to make subroutine calls compact and efficient. The single instruction

```
invokestatic index
```

can be used to perform steps 1-3. `index` is an index into the **JVM constant pool** that represents the static subroutine (class method) being called. Information on the subroutine's frame size and starting address is stored in the corresponding method info entry and class loader entry. Frame creation and saving of necessary stack information are all included as part of the instruction.

A return instruction is used to return from a void subroutine. Steps 5-7 are all included in the return instruction's operation. For functions, a typed return (`areturn`, `dreturn`, `freturn`, `ireturn`, `lreturn`) is used to perform steps 5-7 along with pushing a return value onto the caller's stack (after the subroutine's frame has been popped).

13.2.2 Parameters

In Java both objects and primitive values (integers, floats, etc.) may be passed as parameters. Both kinds of values are passed through the stack. The actual bit pattern of a primitive value is pushed, in one or two words. Objects are passed by pushing a one-word reference (pointer) to the object.

To call a subroutine with n parameters, we evaluate and push each parameter in turn, then call the subroutine (using an `invokestatic` instruction if the subroutine is a static member of a class).

During execution of the subroutine parameters are accessed as locals, starting with index 0. Names of class members (fields and methods) are referenced through references to the constant pool of the class. Hence a reference to #5 in JVM code denotes the fifth entry in the constant pool.

As an example, given the class method

```
static int subr(int a, int b) {return a+b;}
```

and the call `i = subr(1,10)`; we would generate

```
Method int subr(int,int)
```

```
iload          0 ; Push local 0, which is a
```

```

        iload          1 ; Push local 1, which is b
        iadd           ; Compute a+b onto stack
        ireturn        ; Return a+b to caller
; Now the call and assignment
        iconst_1       ; Push 1 as first parm
        bipush         10 ; Push 10 as second parm
        invokestatic   #4 ; Call Method subr
        istore         1 ; Store subr(1,10) in local #1 (i)

```

This approach to parameter passing works for any number of actual parameters. It also readily supports nested calls such as $f(1, g(2))$. As shown below, we begin by pushing f 's first parameter, which is safely protected in the caller's stack. Then g 's parameter is passed and g is called. When g returns, its return value has been pushed on the stack, exactly where f 's second parameter is expected. f can therefore be immediately called. The code is

```

        iconst_1       ; Push f's first parm
        iconst_2       ; Push g's first parm
        invokestatic   #5 ; Call Method g
        invokestatic   #4 ; Call Method f

```

On register-based architectures nested calls are more difficult to implement. Parameter registers already loaded for an enclosing call may need to be reloaded with parameter values for an inner call (see Exercise 12).

13.2.3 Member and Virtual Functions

Calling Instance Methods We have considered how static member functions are called. Now let's look at functions that are members of class instances. Since there can be an unlimited number of instances of a class (i.e., objects), a member of a class instance must have access to the particular object to which it belongs.

Conceptually, each object has its own copy of all the class's methods. Since methods can be very large in size, creating multiple copies of them is wasteful of space. An alternative is to create only one copy of the code for any method. When the method is invoked, it is passed, as an additional invisible parameter, a reference to the object in which the method is to operate.

As an example, consider the following simple class

```
class test {  
    int a;  
    int subr(int b) { return a+b;}  
}
```

To implement the call `tt.subr(2)` (where `tt` is a reference to an object of class `test`), we pass a reference to `tt` as an implicit parameter to `subr`, along with the explicit parameter (2). We also use the instruction `invokevirtual` to call the method.

`invokevirtual` is similar to `invokestatic` except for the fact that it expects an implicit initial parameter that is a reference to the object within which the method will execute. During execution this reference is local 0, with explicit parameters indexed starting at 1. For method `subr` and the call to it we generate

```
Method int subr(int)
```

```

    aload    0          ; Load this pointer from local 0
    getfield #1          ; Load Field this.a
    iload    1          ; Load parameter b (local 1)
    iadd     ; Compute this.a + b
    ireturn  ; Return this.a + b

; Code for the call of tt.subr(2)

    aload           5 ; Push reference to object tt
    iconst_2       ; Push 2
    invokevirtual  #7 ; Call Method test.subr

```

Virtual Functions In Java all subroutines are members of classes. Classes are specially designed to support **subclassing**. A new subclass may be derived from a given class (unless it is final). A class derived from a parent class (a superclass) inherits its parent's fields and methods. The subclass may also define new fields and methods and also redefine existing methods. When an existing method, *m*, is redefined it is often important that all references to *m* use the new definition, even if the calls appear in methods inherited from a parent. Consider

```

class C {
    String myClass() {return "class C";}
    String whoAmI() {return "I am " + myClass();}
}

class D extends C {
    String myClass() {return "class D";}
}

```



```

}
class virtualTest {
    void test() {
        C c_obj = new D();
        System.out.println(c_obj.whoAmI());
    }
}

```

When `c_obj.whoAmI()` is called, the call to `myClass` is resolved to be the definition of `myClass` in class `D`, even though `c_obj` is declared to be of class `C` and a definition of `myClass` exists in class `C` (where `whoAmI` is defined). Methods that are automatically replaced by new definitions in subclasses are called **virtual functions** (or **virtual methods**).

How does a **Java** compiler ensure that the correct definition of a virtual function is selected? In **Java** all instance methods are automatically virtual. The instruction `invokevirtual` uses the class of the object reference passed as argument zero to determine which method definition to use. That is, inside `whoAmI()` the call to `myClass()` generates

```

aload_0          ; Push this, a reference to dobj
invokevirtual #11 ; Call Method myClass() using this ptr

```

The call is resolved inside `dobj`, whose class is `D`. This guarantees that `D`'s definition of `myClass()` is used.

In **C++** member functions may be explicitly designated as virtual. Since **C++** implementations don't generate JVM code, an alternative implementation is used. For classes that contain functions declared virtual, class instances contain a

pointer to the appropriate virtual function to use. Thus in the above example objects of class C would contain a pointer to C's definition of myClass whereas objects of class D contain a pointer to D's definition. When `dobj.whoAmI()` is called, `whoAmI` is passed a pointer to `dobj` (the `this` pointer) which ensures that `whoAmI` will use D's definition of `myClass`.

Static class members are resolved statically. Given

```
class CC {
    static String myClass() {return "class CC";}
    static String whoAmI() {return "I am " + myClass();}
}

class DD extends CC {
    static String myClass() {return "class DD";}
}

class stat {
    static void test() {
        System.out.println(DD.whoAmI());
    }
}
```

"I am class CC" is printed out. The reason for this is that calls to static members use the `invokestatic` instruction which resolves methods using the class of the object within which the call appears. Hence since the call to `myClass` is within class C, C's definition is used. This form of static resolution is exactly the same as that found in C++ when non-virtual member functions are called.

13.2.4 Optimizing Calls

Subroutines are extensively used in virtually all programming languages. Calls can be costly, involving passing parameters, object pointers and return addresses, as well as pushing and popping frames. Hence ways of optimizing calls are of great interest. We shall explore a variety of important approaches.

Inlining CallsSubroutines are normally compiled as **closed subroutines**. That is, one translation of the subroutine is generated, and all calls to it use the same code. An alternative is to translate a call as an **open subroutine**. At the point of call the subroutine body is “opened up,” or expanded. This process is often called **inlining** as the subroutine body is expanded “inline” at the point of call.

In **Java**, inlining is usually restricted to static or final methods (since these can’t be redefined in subclasses). In **C** and **C++** non-virtual functions may be inlined.

Inlining is certainly not suitable for all calls, but in some circumstances it can be a valuable translation scheme. For example, if a subroutine is called at only one place, inlining the subroutine body makes sense. After all, the subroutine body has to appear *somewhere*, so why not at the only place it is used? Moreover, with inlining the expense of passing parameters and pushing and popping a frame can be avoided.

Another circumstance where inlining is widely used is when the body of a subroutine is very small, and the cost of doing a call is as much (or more) than executing the body itself.

It is often the case that one or more parameters to a call are literals. With inlining these literals can be expanded within the subroutine body. This allows simplification of the generated code, sometimes even reducing a call to a single constant value.

On the other hand, inlining large subroutines that are called from many places probably doesn't make sense. Inlining a recursive subroutine can be disastrous, leading to unlimited growth as nested calls are repeatedly expanded. Inlining that significantly increases the overall size of a program may be undesirable, requiring a larger working set in memory and fitting into the instruction cache less effectively. For mobile programs that may be sent across slow communication channels, even a modest growth in program size, caused by inlining, can be undesirable.

Whether a call is translated in the "normal" manner via a subroutine call to a closed body or whether it is inlined, it is essential that the call have the same semantics. That is, since inlining is an optimization it may not affect the results a program computes.

To inline a subroutine call, we enclose the call within a new block. Local variables, corresponding to the subroutine's formal parameters are declared. These local variables are initialized to the actual parameters at the point of call, taking care that the names of formal and actual parameter names don't clash. Then the AST for the subroutine body is translated at the point of call. For functions, a local variable corresponding to the function's return value is also created. A

return involving an expression is translated as an assignment of the expression to the return variable followed by a break from the subroutine body.

As an example, consider

```
static int min(int a, int b) {  
    if (a<=b) return a;  
    else return b;  
}
```

and the call `a = min(1, 3)`. This call is an excellent candidate for inlining as the function's body is very small and the actual parameters are literals. The first step of the inlining is to (in effect) transform the call into

```
{  int parmA = 1;  
    int parmB = 3;  
    int result;  
body:{  
    if (parmA <= parmB) {result = parmA; break body;}  
    else                {result = parmB; break body;}  
    }  
    a=result;  
}
```

Now as this generated code is translated, important simplifications are possible. Using **constant propagation** (see Chapter 16), a variable known to contain a constant value can be replaced by that constant value. This yields

```
{  int parmA = 1;
```

```

    int parmB = 3;
    int result;
body:{
    if (1<=3)  {result = 1; break body;}
    else      {result = 3; break body;}
}
a=result;
}

```

Now $1 \leq 3$ can be simplified (folded) into `true`, and `if (true) ...` can be simplified to its then part. This yields

```

{  int parmA = 1;
   int parmB = 3;
   int result;
body:{
    {result = 1; break body;}
}
a=result;
}

```

Now a break at the very end of a statement list can be deleted as unnecessary, and `result`'s constant value can be propagated to its uses. We now obtain

```

{  int parmA = 1;
   int parmB = 3;
   int result;

```

```
body:{  
    {result = 1;}  
}  
a=1;  
}
```

Finally, any value that is never used within its scope is **dead**; its declaration and definitions can be eliminated. This leads to the final form of our inlined call:

```
a=1;
```

Non-recursive and Leaf Procedures Much of the expense of calls lies in the need to push and pop frames. Frames in general are needed because a subroutine may be directly or indirectly recursive. If a subroutine is recursive, we need a frame to hold distinct values of parameters and locals for each call.

But from experience we know many subroutines *aren't* recursive, and for non-recursive subroutines we can avoid pushing and popping a frame, thereby making calls more efficient. Instead of using a frame to hold parameters and locals, we can statically allocate space for them.

The **Java JVM** is designed around frames. It pushes and pops them as an integral part of call instructions. Moreover, locals may be loaded from a frame very quickly and compactly. However, for most other computer architectures pushing and popping a frame can involve a significant overhead. When a subroutine is known to be non-recursive, all that is needed to do a call is to pass parameters (in

registers when possible) and to do a “subroutine call” instruction that saves the return address and transfers control to the start of the subroutine.

Sometimes a subroutine not only is non-recursive, but it also calls no subroutines at all. Such subroutines are called **leaf procedures**. Again, leaf procedures need no frames. Moreover, since they do no calls at all, neither parameter registers nor the return address register need be protected. This makes calling leaf procedures particularly inexpensive.

How do we determine whether a subroutine is non-recursive? One simple way is to build a **call graph** by traversing the AST for a program. In a call graph, nodes represent procedures, and arcs represent calls between procedures. That is, an arc from node A to node B means a call of B appears in A. After the call graph is built, we can analyze it. If a node A has a path to itself, then A is potentially recursive; otherwise it is not.

Testing for leaf procedures is even easier. If the body of a method contains no calls (that is, no `invokestatic` or `invokevirtual` instructions), then the method is a leaf procedure

Knowing that a procedure is non-recursive or a leaf procedure is also useful in deciding whether to inline calls of it.

Run-time Optimization of Bytecodes Subroutine calls that use the `invokestatic` and `invokevirtual` instructions involve special checks to verify that the methods called exist and are type-correct. The problem is that class definitions may be dynamically loaded, and a call to a method `C.M` that was valid when the call was

compiled may no longer be valid due to later changes to class *C*. When a reference to *C.M* is made in an `invokestatic` or `invokevirtual` instruction, class *C* is checked to verify that *M* still exists and still has the expected type. While this check is needed the first time *C.M* is called, it is wasted effort on subsequent calls. The JVM can recognize this, and during execution, valid `invokestatic` and `invokevirtual` instructions may be replaced with the variants `invokestatic_quick` and `invokevirtual_quick`. These instructions have the same semantics as the original instructions, but skip the unnecessary validity checks after their first execution. Other JVM instructions, like those that access fields, also have quick variants to speed execution.

13.2.5 Higher Order Functions and Closures

Java does not allow subroutines to be manipulated like ordinary data (though a Java superset, *Pizza* [Odersky and Wadler 1997] does). That is, a subroutine cannot be an argument to a call and cannot be returned as the value of a function. However, other programming languages, like *ML* [Milner et al 1997] and *Haskell* [Jones et al 1998], are **functional** in nature. They encourage the view that functions are just a form of data that can be freely created and manipulated. Functions that take other functions as parameters or return parameters as results are called **higher-order**.

Higher-order functions can be quite useful. For example, it is sometimes the case that computation of $f(x)$ takes a significant amount of time. Once $f(x)$ is known, it is a common optimization, called **memoizing**, to table the pair

$(x, f(x))$ so that subsequent calls to f with argument x can use the known value of $f(x)$ rather than recompute it. In ML it is possible to write a function `memo` that takes a function f and an argument `arg`. The function `memo` computes $f(\text{arg})$ and also returns a “smarter” version of f that has the value of $f(\text{arg})$ “built into” it. This smarter version of f can be used instead of f in all subsequent computations.

```
fun memo(fct,parm)= let val ans = fct(parm) in
  (ans, fn x=> if x=parm then ans else fct(x))end;
```

When the version of `fct` returned by `memo` is called, it must have access to the values of `parm`, `fct` and `ans`, which are used in its definition. After `memo` returns, its frame must be preserved since that frame contains `parm`, `fct` and `ans` within it.

In general in languages with higher-order functions, when a function is created or manipulated, we must maintain a pair of pointers. One is to the machine instructions that implement the function, and the other is to the frame (or frames) that represent the function’s **execution environment**. This pair of pointers is called a **closure**. When functions are higher-order, a frame corresponding to a call may be accessed *after* the call terminates (this is the case in `memo`). Thus frames can’t always be allocated on the run-time stack. Rather, they are allocated in the heap and garbage-collected, just like user-created data. This appears to be inefficient, but it need not be if an efficient garbage collector is used (see Section 11.3).

13.2.6 Exception Handling

Java, like most other modern programming languages, provides an **exception handling** mechanism. During execution, an exception may be **thrown**, either explicitly (via a throw statement) or implicitly (due to an execution error). Thrown exceptions may be **caught** by an **exception handler**. When an exception is thrown control transfers immediately to the catch clause that handles it. Code at the point where the throw occurred is not resumed.

Exceptions form a clean and general mechanism for identifying and handling unexpected or erroneous situations. They are cleaner and more efficient than using error flags or gotos. Though we will focus on Java's exception handling mechanism, most recent language designs, including C++, Ada and ML, include an exception handling mechanism similar to that of Java.

Java exceptions are **typed**. An exception throws an object that is an instance of class `Throwable` or one of its subclasses. The object thrown may contain fields that characterize the precise nature of the problem the exception represents, or the class may be empty (with its type signifying all necessary information).

As an example consider the following Java code

```
class NullSort extends Throwable{};

    int  b[] = new int[n];

    try {

        if (b.length > 0)

            // sort b
```

```
        else throw new NullSort(); }  
    catch (NullSort ns) {  
        System.out.println("Attempt to sort empty array");  
    }  
}
```

An integer array `b` is created and it is sorted within the try clause. The length of `b` is tested and a `NullSort` exception is thrown if the array's length is 0 (since trying to sort an empty array may indicate an error situation.) In the example, when a `NullSort` exception is caught, an error message is printed.

Exceptions are designed to be **propagated dynamically**. If a given exception isn't caught in the subroutine (method) where it is thrown, it is propagated back to the caller. In particular, a return is effected (popping the subroutine's frame) and the exception is rethrown at the return point. This process is repeated until a handler is found (possibly a default handler at the outermost program level).

In practice, we'd probably package the `sort` as a subroutine (method), allowing it to throw and propagate a `NullSort` exception:

```
...  
static int [] sort(int [] a) throws NullSort {  
    if (a.length > 0)  
        // sort a  
    else throw new NullSort();  
    return a;  
}  
...
```

```
int b[] = new int[n];  
try {sort(b);}   
catch (NullSort ns) {  
    System.out.println("Attempt to sort empty array");  
} ...
```

Exception handling is straightforward in Java. The JVM maintains an **exception table** for each method. Each table entry contains four items: **fromAdr**, **toAdr**, **exceptionClass** and **handlerAdr**. **fromAdr** is the first bytecode covered by this entry; **toAdr** is the last bytecode covered. **exceptionClass** is the class of exception handled. **exceptionAdr** is the address of the corresponding handler. If an exception is thrown by an instruction in the range **fromAdr..toAdr**, the class of the thrown exception is checked. If the thrown class **matches exceptionClass** (equals the class or is a subclass of it), control passes to the **exceptionAdr**. If the class thrown doesn't match **exceptionClass** or the exception was thrown by an instruction outside **fromAdr..toAdr**, the next item in the exception table is checked. If the thrown exception isn't handled by any entry in the exception table, the current method is forced to return and the exception is thrown again at the return point of the call.

Since all this checking is automatically handled as part of the execution of the JVM, all we need to do is create exception table entries when try and catch blocks are compiled. In particular, we generate labels to delimit the extent of a try block, and pass these labels to catch blocks, which actually generate the exception table entries.

On architectures other than the JVM, exception processing is more difficult. Code must be generated to determine which exception handler will process an exception of a given class thrown from a particular address. If no handler is defined, a return must be effected (popping the stack, restoring registers, etc.). Then the exception is rethrown at the call's return address.

We'll begin with the translation of a `tryNode`, whose structure is shown in Figure 13.17. A try statement may have a finally clause that is executed whenever

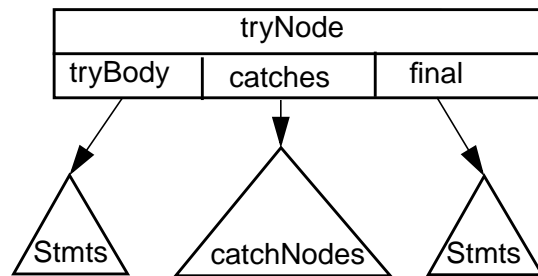


Figure 13.17 Abstract Syntax Tree for a Try Statement

a try statement is exited. A finally clause complicates exception handling, so we'll defer it until the next section. For now, we'll assume `final` is null. In fact, finally clauses are rarely used, so we will be considering the most common form of exception handling in Java.

First, we will define a few utility and code generation subroutines. `GenStore(identNode)` will generate code to store the value at the stack top into the identifier represented by `identNode`. `addToExceptionTable(fromLabel, toLabel, exceptionClass, catchLabel)` will add the four tuple (`fromLabel`, `toLabel`, `exceptionClass`, `catchLabel`) to current exception table.

```

tryNode.CodeGen( )
1.  fromLabel ← CreateUniqueLabel()
2.  GenLabel(fromLabel)
3.  tryBody.CodeGen()
4.  exitLabel ← CreateUniqueLabel()
5.  GenGoTo(exitLabel)
6.  toLabel ← CreateUniqueLabel()
7.  GenLabel(toLabel)
8.  catches.CodeGen(fromLabel, toLabel, exitLabel)
9.  GenLabel(exitLabel)

```

Figure 13.18 Code Generation Routine for Try Statements

In translating a try statement, we will first generate assembly-level labels immediately before and immediately after the code generated for `tryBody`, the body of the try block. `catches`, a list of `catchNodes` is translated after the try block. We generate a branch around the code for catch blocks as they are reached only by the JVM's exception handling mechanism. This translation is detailed in Figure 13.18.

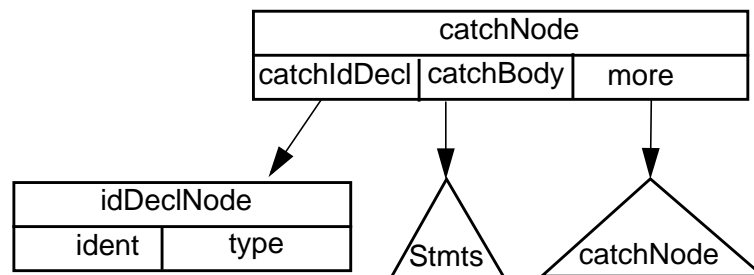


Figure 13.19 Abstract Syntax Tree for a Catch Block

A `catchNode` is shown in Figure 13.19. To translate a catch block we first define a label for the catch block. When the catch block is entered, the object thrown is on the stack. We store it into the exception parameter `ident`. We translate `catchBody`, and then add an entry to the end of the current method's `except-`

tion table. Finally, more, the remaining catch blocks, are translated. This translation is detailed in Figure 13.20.

```

    catchNode.CodeGen( fromLabel, toLabel, exitLabel )
1.  catchLabel ← CreateUniqueLabel()
2.  GenLabel(catchLabel)
3.  catchIdDecl.CodeGen()
4.  GenStore(catchIdDecl.Ident)
5.  catchBody.CodeGen()
6.  addToExceptionTable(fromLabel, toLabel-1, catchIdDecl.Type,
                        catchLabel)
7.  if more ≠ null
8.      then GenGoTo(exitLabel)
9.      more.CodeGen(fromLabel, toLabel, exitLabel)

```

Figure 13.20 Code Generation Routine for Catch Blocks

Since exception table entries are added to the end of the table, our code generator correctly handles nested try blocks and multiple catch blocks.

As an example, consider the following Java program fragment. In the try block, $f(a)$ is called; it may throw an exception. If no exception is thrown, b is assigned the value of $f(a)$. If exception e_1 is thrown, b is assigned 0. If e_2 is thrown, b is assigned 1. If any other exception is thrown, the exception is propagated to the containing code. In the exception table, the expression $L-1$ denotes one less than the address assigned to L .

```

class e1 extends Throwable{};
class e2 extends Throwable{};

try {b = f (a);}

catch (e1 v1) {b = 0;}

catch (e2 v2) {b = 1;}

```

The code we generate is


```

L1: iload          2 ; Push local #2 (a) onto stack
    invokestatic  #7 ; Call Method f
    istore        1 ; Store f(a) into local #1 (b)
    goto          L2 ; Branch around catch blocks
L3:                ; End of try block
L4: astore        3 ; Store thrown obj in local #3 (v1)
    iconst_0      ; Push 0
    istore        1 ; Store 0 into b
    goto          L2 ; Branch around other catch block
L5: astore        4 ; Store thrown obj in local #4 (v2)
    iconst_1      ; Push 1
    istore        1 ; Store 1 into b

L2:

```

Exception table:

from	to	catch	exception
L1	L3-1	L4	Class e1
L1	L3-1	L5	Class e2

Finally blocks and Finalization We exit a block of statements after the last statement of the block is executed. This is **normal** termination. However, a block may be

exited **abnormally** in a variety of ways, including executing a break or continue, doing a return, or throwing an exception.

Prior to exiting a block, it may be necessary to execute **finalization code**. For example in C++, destructors for locally allocated objects may need to be executed. If reference counting is used to collect garbage, counts must be decremented for objects referenced by local pointers.

In Java, finalization is performed by a finally block that may be appended to a try block. The statements in a finally block must be executed no matter how the try block is exited. This includes normal completion, abnormal completion because of an exception (whether caught or not) and premature completion because of a break, continue or return statement.

Since the finally block can be reached in many ways, it is not simply branched to. Instead it is reached via a simplified subroutine call instruction, the `jsr`. This instruction branches to the subroutine and saves the return address on the stack. No frame is pushed. After the finally statements are executed, a `ret` instruction returns to the call point (again no frame is popped).

A try block that exits normally will “call” the finally block just before the entire statement is exited. A catch block that handles an exception thrown in the try block will execute its body and then call the finally block. An exception not handled in any catch block will have a default handler that calls the finally block and then rethrows the exception for handling by the caller. A break, continue or exit will call the finally block before it transfers control outside the try statement.

```

tryNode.CodeGen( )
1.  fromLabel ← CreateUniqueLabel()
2.  GenLabel(fromLabel)
3.  if final ≠ null
4.      then finalLabel ← CreateUniqueLabel()
5.          finalList ← listNode(finalLabel, finalList)
6.  tryBody.CodeGen()
7.  if final ≠ null
8.      then GenJumpSubr(finalLabel)
9.          finalList ← finalList.next
10.     else finalLabel ← null
11.  exitLabel ← CreateUniqueLabel()
12.  GenGoTo(exitLabel)
13.  toLabel ← CreateUniqueLabel()
14.  GenLabel(toLabel)
15.  catches.CodeGen(fromLabel, toLabel, exitLabel, finalLabel)
16.  if final ≠ null
17.      then defaultHandlerLabel ← CreateUniqueLabel()
18.          GenLabel(defaultHandlerLabel)
19.          exceptionLoc ← GenLocalDecl(Throwable)
20.          GenStoreLocalObject(exceptionLoc)
21.          GenJumpSubr(finalLabel)
22.          GenLoadLocalObject(exceptionLoc)
23.          GenThrow()
24.          addToExceptionTable(fromLabel, defaultHandlerLabel-1,
                               Throwable, defaultHandlerLabel)
25.          GenLabel(finalLabel)
26.          returnLoc ← GenLocalDecl(Address)
27.          GenStoreLocalObject(returnLoc)
28.          final.CodeGen()
29.          GenRet(returnLoc)
30.  GenLabel(exitLabel)

```

Figure 13.21 Code Generation Routine for Try Statements with Finally blocks

We will consider the complications finally blocks add to the translation of a `tryNode` and a `catchNode`. First, we will define a few more utility and code generation subroutines. We'll again use `GenLocalDecl(Type)`, which declares a local variable and returns its frame offset or variable index.

```

    catchNode.CodeGen( fromLabel, toLabel, exitLabel, finalLabel )
1.  catchLabel ← CreateUniqueLabel()
2.  GenLabel(catchLabel)
3.  catchIdDecl.CodeGen()
4.  GenStore(catchIdDecl.Ident)
5.  catchBody.CodeGen()
6.  if finalLabel ≠ null
7.      then GenJumpSubr(finalLabel)
8.  addToExceptionTable(fromLabel, toLabel-1, catchIdDecl.Type,
                        catchLabel)
9.  if more ≠ null or finalLabel ≠ null
10.     then GenGoTo(exitLabel)
11.  if more ≠ null
12.     then more.CodeGen(fromLabel, toLabel, exitLabel, finalLabel)

```

Figure 13.22 Code Generation Routine for Catch Blocks with Finally Blocks

Recall that `GenJumpSubr(Label)` generates a subroutine jump to `Label` (a `jsr` in the JVM). `GenRet(localIndex)` will generate a return using the return address stored at `localIndex` in the current frame (in the JVM this is a `ret` instruction). `GenThrow()` will generate a throw of the object currently referenced by the top of the stack (an `athrow` instruction in the JVM).

`GenStoreLocalObject(localIndex)` will store an object reference into the frame at `localIndex` (in the JVM this is an `astore` instruction). `GenLoadLocalObject(localIndex)` will load an object reference from the frame at `localIndex` (in the JVM this is an `aload` instruction).

An extended code generator for a `tryNode` that can handle a finally block is shown in Figure 13.21. A non-null `final` field requires several additional code generation steps. First, in lines 4 and 5, a label for the statements in the finally block is created and added to `finalList` (so that `break`, `continue` and `return` statements can find the finally statements that need to be executed).

In line 8, a subroutine jump to the finally block is placed immediately after the translated try statements (so that finally statements are executed after statements in the try block). In line 9 the current `finalLabel` is removed from the `finalList` (because the `tryBody` has been completely translated).

After all the `catchNodes` have been translated, several more steps are needed. In lines 17 to 24 a default exception handler is created to catch all exceptions not caught by user-defined catch blocks. The default handler stores the thrown exception, does a subroutine jump to the finally block, and then rethrows the exception so that it can be handled by an enclosing handler.

In lines 25 to 29, the finally block is translated. Code to store a return address in a local variable is generated, the statements within the finally block are translated, and a return is generated.

An extended code generator for a `catchNode` that can handle a finally block is shown in Figure 13.22. A `finalLabel` parameter is added. In line 7 a subroutine call to the `finalLabel` is added after the statements of each `catchBody`.

As an example, consider the translation of the following try block with an associated finally block.

```
class e1 extends Throwable{};
try {a = f(a);}
catch (e1 v1) {a = 1;}
finally {b = a;}
```

The code we generate is

```
L1: iload          2 ; Push a onto stack
```

```

        invokestatic  #7 ; Call Method f
        istore       2  ; Store f(a) into local #2 (a)
        jsr         L2 ; Execute finally block
        goto        L3 ; Branch past catch & finally block
L4:
        ; End of try block
L5:
        astore      3  ; Store thrown obj in local #3 (v1)
        iconst_1    ; Push 1
        istore      2  ; Store 1 into a
        jsr         L2 ; Execute finally block
        goto        L3 ; Branch past finally block
; Propagate uncaught exceptions
L6:
        astore      4  ; Store thrown obj into local #4
        jsr         L2 ; Execute finally block
        aload       4  ; Reload thrown obj from local #4
        athrow      ; Propagate exception to caller
L2: astore      5  ; Store return adr into local #5
        iload       2  ; Load a
        istore      1  ; Store a into local #1 (b)
        ret         5  ; return using adr in local #5
L3:
Exception table:

```

```
from    to    catch  exception
L1      L4-1  L5      Class e1
L1      L6-1  L6      Class Throwable
```

13.2.7 Support for Run-Time Debugging

Most modern programming languages are supported by a sophisticated symbolic debugger. Using this tool, programmers are able to watch a program and its variables during execution. It is essential, of course, that a debugger and compiler effectively cooperate. The debugger must have access to a wide variety of information about a program, including the symbolic names of variables, fields and subroutines, the run-time location and values of variables, and the source statements corresponding to run-time instructions.

The Java JVM provides much of this information automatically, as part of its design. Field, method, and class names are maintained, in symbolic form, in the run-time class file, to support dynamic linking of classes. Attributes of the class file include information about source files, line numbers, and local variables. For other languages, like C and C++, this extra information is added to a standard “.o” file, when debugging support is requested during compilation.

Most of the commands provided by a debugger are fairly easy to implement given the necessary compiler support. Programs may be executed a line at a time by extracting the run-time instructions corresponding to a single source line and

executing them in isolation. Variables may be examined and updated by using information provided by a compiler on the variable's address or frame offset.

Other commands, like setting breakpoints, take a bit more care. A JVM `breakpoint` instruction may be inserted at a particular line number or method header. Program code is executed normally (to speed execution) until a breakpoint is executed. Normal execution is then suspended and control is returned to the debugger to allow user interaction. `breakpoint` instructions may also be used to implement more sophisticated debugging commands like "watching" a variable. Using compiler-generated information on where variables and fields are updated, `breakpoint` instructions can be added wherever a particular variable or field is changed, allowing its value to be displayed to a user whenever it is updated.

Optimizations can significantly complicate the implementation of a debugger. The problem is that an optimizer often changes how a computation is performed to speed or simplify it. This means what is actually computed at run-time may not correspond exactly to what a source program seems to specify. For example, consider the statements

```
a = b + c;
```

```
d = a + 1;
```

```
a = 0;
```

We might generate

```
iload      1  ; Push b onto stack
```

```
iload      2  ; Push c onto stack
```



```
iadd          ; Compute a = b + c
iconst_1     ; Push 1
iadd          ; Compute a + 1
istore       3 ; Store d
iconst_0     ; Push 0
istore       0 ; Store a
```

`a` is computed in the first assignment statement, used in the second and reassigned in the third. Rather than store `a`'s value and then immediately reload it, the generated code uses the value of `a` computed in the first assignment without storing it. Instead, the second, final value of `a` is stored.

The problem is that if we are debugging this code and ask to see `a`'s value immediately after the second statement is executed, we'll not see the effect of the first assignment (which was optimized away). Special care is needed to guarantee that the "expected" value is seen at each point, even if that value is eliminated as unnecessary!

For a more thorough discussion of the problems involved in debugging optimized code see [Adl-Tabatabai and Gross 1996].

Exercises

1. Since many programs contain a sequence of if-then statements, some languages (like Ada) have extended if statements to include an **elsif** clause:

```
if expr1
    stmt1
elsif expr2
    stmt2
elsif expr3
    stmt3
...
else stmtn
```

Each of the expressions is evaluated in turn. As soon as a true expression is reached, the corresponding statements are executed and the if-elsif statement is exited. If no expression evaluates to true, the else statement is executed.

Suggest an AST structure suitable for representing an if-elsif statement. Define a code generator for this AST structure. (You may generate JVM code or code for any other computer architecture).

2. Assume that we create a new kind of conditional statement, the **signtest**. Its structure is

```
signtest expression
    neg: statements
    zero: statements
    pos: statements
```

```
end
```

The integer-valued `expression` is evaluated. If it is negative, the statements following `neg` are executed. If it is zero, the statements following `zero` are executed. If it is positive, the statements following `pos` are executed.

Suggest an AST structure suitable for representing a `signtest` statement. Define a code generator for this AST structure. (You may generate JVM code or code for any other computer architecture).

3. Assume we add a new kind of looping statement, the `exit-when` loop. This loop is of the form

```
loop
    statements1
    exit when expression
    statements2
end
```

First `statements1` are executed. Then `expression` is evaluated. If it is true, the loop is exited. Otherwise, `statements2` followed by `statements1` are executed. Then `expression` is reevaluated, and the loop is conditionally exited. This process repeats until `expression` eventually becomes true (or else the loop iterates forever).

Suggest an AST structure suitable for representing an `exit-when` loop. Define a code generator for this AST structure. (You may generate JVM code or code for any other computer architecture).

4. In most cases switch statements generate a jump table. However, when the range of case labels is very sparse, a jump table may be too large. A search table, if sorted, can be searched using a logarithmic time binary search rather than a less efficient linear search

If the number of case labels to be checked is small (less than 8 or so), the binary search can be expanded into a series of nested if statements. For example, given

```
switch(i) {  
    case -1000000:  a = 1; break;  
    case 0:        a = 3; break;  
    case 1000000:  a = 5; break;  
}
```

we can generate

```
if (i <= 0)  
    if (i == -1000000)  
        a = 1;  
    else    a = 3;  
else      a = 5;
```

Explain why this “expanded” binary search can be more efficient than doing a binary search of a list of case label, assembly label pairs.

How would you change the code generator for switch statements to handle this new translation scheme?

5. Some languages, like *Ada*, allow switch statements to have cases labeled with a range of values. For example, with ranges, we might have the following in *Java* or *C*.

```
switch (j) {  
    case 1..10,20,30..35 : option = 1; break;  
    case 11,13,15,21..29 : option = 2; break;  
    case 14,16,36..50    : option = 3; break;  
}
```

How would you change the code generator of Figure 13.15 to allow case label ranges?

6. Switch and case statements normally require that the control expression be of type integer. Assume we wish to generalize a switch statement to allow control expressions and case labels that are floating point values. For example, we might have

```
switch(sqrt(i)) {  
    case 1.4142135: val = 2; break;  
    case 1.7320508: val = 3; break;  
    case 2:        val = 4; break;  
    default:      val = -1;  
}
```

Would you recommend a jump table translation, or a search table translation, or is some new approach required? How would you handle the fact that float-

ing point equality comparisons are “fuzzy” due to roundoff errors. (That is, sometimes what should be an identity relation, like `sqrt(f)*sqrt(f) == f` is *false*.)

7. In Section 13.2.1 we listed seven steps needed to execute a parameterless call. Assume you are generating code for a register-oriented machine like the Mips or Sparc or x86 (your choice). Show the instructions that would be needed to implement each of these seven steps.

Now look at the code generated for a parameterless call by a C or C++ compiler on the machine you selected. Can you identify each of the seven steps needed to perform the call?

8. Consider the following C function and call

```
char select(int i, char *charArray)
    return charArray[i];
c = select(3, "AbCdEfG");
```

Look at the code generated for this function and call on your favorite C or C++ compiler. Explain how the generated code passes the integer and character array parameters and how the return value is passed back to the caller.

9. Many register-oriented computer architectures partition the register file(s) into two classes, **caller-save** and **callee-save**. Caller-save registers in use at a call site must be explicitly saved prior to the call and restored after the call (since they may be freely used by the subroutine that is to be called). Callee-save registers that will be used within a subroutine must be saved by the sub-

routine (prior to executing the subroutine's body) and restored by the subroutine (prior to returning to the caller). By allocating caller-save and callee-save registers carefully, fewer registers may need to be saved across a call.

Extend the seven steps of Section 13.2.1 to provide for saving and restoring of both caller-save and callee-save registers.

10. Assume we know that a subroutine is a leaf procedure (i.e., that it contains no calls within its body). If we wish to allocate local variables of the subroutine to registers, is it better to use caller-save or callee-save registers? Why?
11. A number of register-oriented architectures require that parameters be placed in particular registers prior to a call. Thus the first parameter may need to be loaded into register `parm1`, the second parameter into register `parm2`, etc. Obviously, we can simply evaluate parameter expressions and move their values into the required register just prior to beginning a call. An alternative approach involves **register targeting**. That is, prior to evaluating a parameter expression, the desired target register is set and code generators for expressions strive to evaluate the expression value directly into the specified register.

Explain how the **Result** mechanism of Chapter 12 can be used to effectively implement targeting of parameter values into designated parameter registers.

12. As we noted in Section 13.2.2, nested calls are easy to translate for the JVM. Parameter values are simply pushed onto the stack, and used when all required parameters are evaluated and pushed.

However, matters become more complex when parameter registers are used.

Consider the following call

```
a = p(1, 2, p(3, 4, 5));
```

Note that parameter registers loaded for one call may need to be saved (and later reloaded) if an “inner call” is encountered while evaluating a parameter expression.

What extra steps must be added to the translation of a function call if the call appears within an expression that is used as a parameter value?

13. We have learned that some machines (like the JVM) pass parameters on the run-time stack. Others pass parameters in registers. Still another approach (a very old one) involves placing parameter values in the program text, immediately after the subroutine call instruction. For example $f(1, b)$ might be translated into

```
call f
1
value of b
instructions following call of f
```

Now the “return address” passed to the subroutine actually points to the first parameter value, and the actual return point is the return address value + n , where n parameters are passed.

What are the advantages and disadvantages of this style of parameter passing as contrasted with the stack and register-oriented approaches?

14. Assume we have an AST representing a call and that we wish to “inline” the call. How can this AST be rewritten, prior to code generation, so that it represents the body of the subroutine, with actual parameter values properly bound to formal parameters?
15. The Java JVM is unusual in that it provides explicit instructions for throwing and catching exceptions. On most other architectures, these mechanisms must be “synthesized” using conventional instructions.

To throw an exception object O , we must do two things. First, we must decide whether the throw statement appears within a try statement prepared to catch O . If it is, O is passed to the innermost catch block declared to handle O (or one of O 's superclasses).

If no try block that catches O encloses the throw, the exception is propagated—a return is effected and O is rethrown at the return point.

Suggest run-time data structures and appropriate code to efficiently implement a throw statement. Note that ordinary calls should not be unduly slowed just to prepare for a throw that may never occur. That is, most (or all) of the cost of a throw should be paid only after the throw is executed.

16. When a run-time error occurs, some programs print an error message (with a line number), followed by a call trace which lists the sequence of pending subroutine calls at the time the error occurred. For example, we might have:

```
Zero divide error in line 12 in procedure
    "WhosOnFirst"
```

Called from line 34 in procedure "WhatsOnSecond"

Called from line 56 in procedure "IDontKnowsOnThird"

Called from line 78 in procedure "BudAndLou"

Explain what would have to be done to include a call trace (with line numbers) in the code generated by a compiler. Since run-time errors are rare, the solution you adopt should impose little, if any, cost on ordinary calls. All the expense should be borne after the run-time error occurs.

14

Processing Data Structure Declarations and References

14.1 Type Declarations

As we learned in Chapters 8, 9 and 10, type declarations provide much of the information that drives semantic processing and type checking. These declarations also provide information vital for the translation phase of compilation.

Every data object must be allocated space in memory at run-time. Hence it is necessary to determine the space requirements for each kind of data object, whether predefined or user-defined.

We will assume that the symbol table entry associated with each type or class name has a field (an attribute) called `size`. For predefined types `size` is determined by the specification of the programming language. For example, in Java an `int` is assigned a `size` of 4 bytes (one word), a `double` is assigned a `size` of 8 bytes (a double word), etc. Some predefined types, like strings, have their `size` determined by the content of the string. Hence in C or C++ the string "cat" will require 4 bytes (including the string terminator). In Java, where strings are implemented as `String` objects, "cat" will require at least 3 bytes, and probably more (to store the `length` field as well as type and allocation information for the object).

As discussed below, classes, structures and arrays have a size determined by the number and types of their components. In all cases, allocation of a value of type `T` means allocation of a block of memory whose size is `T.size`.

An architecture also imposes certain “natural sizes” to data values that are computed. These sizes are related to the size of registers or stack values. Thus the value of `A+1` will normally be computed as a 32 bit value even if `A` is declared to be a `byte` or `short`. When a value computed in a natural size is stored, truncation may occur.

14.2 Static and Dynamic Allocation

Identifiers can have either global or local scope. A global identifier is visible throughout the entire text of a program. The object it denotes must be allocated throughout the entire execution of a program (since it may be accessed at any time).

In contrast, a local identifier is visible only within a restricted portion of a program, typically a subprogram, block, structure or class. The object a local identifier denotes need only be allocated when its identifier is visible.

We normally distinguish two modes of allocation for run-time variables (and constants)—**static** and **dynamic** allocation. A local variable that exists only during the execution of a subprogram or method can be dynamically allocated when a call is made. Upon return, the variable ceases to exist and its space can be deallocated.

In contrast, a globally visible object exists for the entire execution of a program; it is allocated space statically by the compiler prior to execution.

In C and C++ global variables are declared outside the body of any subprogram. In Java, global variables occur as static members of classes. Static local variables in C and C++ are also given static allocation—only one memory allocation is made for the whole execution of the program.

C, C++ and Java also provide for dynamic allocation of **values** on a run-time heap. Class objects, structures and arrays may be allocated in the heap, but all such values must be accessed in a program through a local or global variable (or constant).

What happens at run-time when a declaration of local or global is executed? First and foremost, a memory location must be allocated. The **address** of this location will be used whenever the variable is accessed.

The details of how a memory location is allocated is machine-dependent, but the general approach is fairly standard. Let’s consider globals first. In C and C++ (and many other related languages like Pascal, Ada and Fortran), a block of run-time memory is created to hold all globals (variables, constants, addresses, etc.). This block is extended as global declarations are processed by the compiler, using the `size` attribute of the variable being declared. For example, a compiler might create a label `globals`, addressing an initially empty block of memory. As global declarations are processed, they are given addresses relative to `globals`. Thus the declarations

```
int a;
char b[10];
float c;
```

would cause `a` to be assigned an address at `globals+0` and `b` to be assigned an address at `globals+4`. We might expect `c` to receive an address equivalent to `globals+14`, but on many machines **alignment** of data must be considered. A `float` is usually one word long, and word-length values often must be placed at memory addresses that are a multiple of 4. Hence `c` may well be placed at an address of `globals+16`, with 2 bytes lost to guarantee proper alignment.

In Java, global variables appear as static members of a program's classes. These members are allocated space within a class when it is loaded. No explicit addresses are used (for portability and security reasons). Rather, a static member is read using a `getstatic` bytecode and written using a `putstatic` bytecode. For example, the globals `a`, `b` and `c`, used above, might be declared as

```
class C {
    static int a;
    static int b[] = new int[10];
    static float f;
}
```

The assignment `C.a = 10;` would generate

```
bipush 10
putstatic C/a I
```

The notation `C/a` resolves to a reference to field `a` of class `C`. The symbol `I` is a **typecode** indicating that field `a` is an `int`. The JVM loader and interpreter translate this reference into the memory location assigned to field `a` in `C`'s class file.

Similarly, the value of `C.f` (a `float` with a typecode of `F`) is pushed onto the top of the run-time stack using

```
getstatic C/f F
```

Locals are dynamically created when a function, procedure or method is called. They reside within the frame created as part of the call (see Section 11.2). Locals are assigned an offset within a frame and are addressed using that offset. Hence if local variable `i` is assigned an offset of 12, it may be addressed as `frame_pointer+12`. On most machines the frame is accessed using a dedicated frame register or using the stack top. Hence if `$fp` denotes the frame register, `i` can be accessed as `$fp+12`. This expression corresponds to the **indexed** addressing mode found on most architectures. Hence a local may be read or written using a single load or store instruction.

On the JVM, locals are assigned indices within a method's frame. Thus local `i` might be assigned index 7. This allows locals to be accessed using this index into the current frame. For example `i` (assuming it is an integer) can be loaded onto the run-time stack using

```
iload 7
```

Similarly, the current stack-top can be stored into `i` using

```
istore 7
```

Corresponding load and store instructions exist for all the primitive types of Java (`fload` to load a float, `astore` to store an object reference, etc.).

14.2.1 Initialization of Variables

Variables may be initialized as part of their declaration. Indeed, Java mandates a default initialization if no user-specified initialization is provided.

Initialization of globals can be done when a program or method is loaded into memory. That is, when the memory block assigned to global variables is created, a bit pattern corresponding to the appropriate initial values is loaded into the memory locations. This is an easy thing to do—memory locations used to hold program code are already initialized to the bit patterns corresponding to the program's machine-level instructions. For more complex initializations that involve variables or expressions, assignment statements are generated at the very beginning of the main program.

Initialization of locals is done primarily through assignment statements generated at the top of the subprogram's body. Thus the declaration

```
int limit = 100;
```

would generate a store of 100 into the location assigned to `limit` prior to execution of the subprogram's body. More complex initializations are handled in the same way, evaluating the initial value and then storing it into the appropriate location within the frame.

14.2.2 Constant Declarations

We have two choices in translating declarations of constant (`final`) values. We can treat a constant declaration just like a variable declaration initialized to the value of the constant. After initialization, the location may be read, but is never written (semantic checking enforces the rule that constants may not be redefined after declaration).

For constants whose value is determined at run-time, we have little choice but to translate the constant just as we would a variable. For example given

```
const int limit = findMax(dataArray);
```

we must evaluate the call to `findMax` and store its result in a memory location assigned to `limit`. Whenever `limit` is read, the value stored in the memory location is used.

Most constants are declared to be **manifest constants** whose value is evident at compile-time. For these declarations we can treat the constant as a synonym for its defining value. This means we can generate code that directly accesses the defining value, bypassing any references to memory. Thus given the Java declaration

```
final int dozen = 12;
```

we can substitute 12 for references to `dozen`. Note however that for more complex constant values it may still be useful to assign a memory location initialized to the defining value. Given

```
final double pi = 3.1415926535897932384626433832795028841971;
```

there is little merit in accessing the value of `pi` in literal form; it is easier to allocate a double word and initialize it once to the appropriate value. Note too that local manifest constants can be allocated globally. This is a useful thing to do since

the overhead of creating and initializing a location need be paid only once, at the very start of execution.

14.3 Classes and Structures

One of the most important data structures in modern programming language is the class (or structure or record). In Java, all data and program declarations appear within classes. In C and C++, structures and classes are widely used.

What must be done to allocate and use a class object or structure? Basically, classes and structures are just containers that hold a collection of fields. Even classes, which may contain method declarations, really contain only fields. The bodies of methods, though logically defined within a class, actually appear as part of the translated executable code of a program.

When we process a class or structure declaration, we allocate space for each of the fields that is declared. Each field is given an offset within the class or structure, and the overall size necessary to contain all the fields is determined. This process is very similar to the assignment of offsets for locals that we discussed above.

Let us return to our earlier example of variables *a*, *b* and *c*, now encapsulated within a C-style structure:

```
struct {
    int a;
    char b[10];
    float f;
} s
```

As each field declaration is processed, it is assigned an offset, starting at 0. Offsets are adjusted upward, if necessary, to meet alignment rules. The overall size of all fields assigned so far is recorded. In this example we have

Field	Size	Offset
a	4	0
b	10	4
c	4	16

The size of *S* is just *c*'s offset plus its size. *S* also has an **alignment requirement** that corresponds to *a*'s alignment requirement. That is, when objects of type *S* are allocated (globally, locally or in the heap), they must be word-aligned because field *a* requires it.

In Java, field offsets and alignment are handled by the JVM and class file structure. Within a class file for *C*, each of its fields is named, along with its type. An overall size requirement is computed from these field specifications.

To access a field within a structure or class, we need to compute the address of the field. The formula is simple

$$\text{address}(A.b) = \text{address}(A) + \text{Offset}(b).$$

That is, the address of each field in a class object or structure is just the starting address of the object plus the field's offset within the object. With static allocation no explicit addition is needed at run-time. If the object has a static address Adr , and b 's offset has value f , then $\text{Adr}+f$, which is a valid address expression, can be used to access $A.b$.

In Java, the computation of a field address is incorporated into the `getField` and `putField` bytecodes. An object reference is first pushed onto the stack (this is the object's address). The instruction

```
getField C/f F
```

determines the offset of field f (a `float` with a typecode of F) in class C , and then adds this offset to the object address on the stack. The data value at the resulting address is then pushed onto the stack. Similarly,

```
putField C/f F
```

fetches the object address and data value at the top of the stack. It adds f 's offset in class C , adds it to the object address, and stores the data value (a `float`) at the field address just computed.

For local class objects or structures, allocated in a frame, we use the same formula. The address of the object or struct is represented as `frame_pointer+ObjectOffset`. The field offset is added to this: `frame_pointer+ObjectOffset+FieldOffset`. Since the object and field offsets are both constants, they can be added together to form the usual address form for locals, `frame_pointer+Offset`. As usual, this address reduces to a simple indexed address within an instruction.

In Java, class objects that are locals are accessed via object references that are stored within a frame. Such references are loaded and stored using the `aload` and `astore` bytecodes. Fields are accessed using `getField` and `putField`.

Heap-allocated class objects or structures are accessed through a pointer or object reference. In Java all objects **must** be heap-allocated. In C, C++ and related languages, class objects and structures may be dynamically created using `malloc` or `new`. In these cases, access is much the same as in Java—a pointer to the object is loaded into a register, and a field in the object is accessed as `register+offset` (again the familiar indexed addressing mode).

14.3.1 Variant Records and Unions

A number of varieties of classes and structures are found in programming languages. We'll discuss two of the most important of these—**variant records** and **unions**.

Variant records were introduced in Pascal. They are also found in other programming languages, including Ada. A variant record is a record (that is, a structure) with a special field called a **tag**. The tag, an enumeration or integer value, chooses among a number of mutually exclusive groups of fields called **variants**. That is, a variant record contains a set of fixed fields plus additional fields that may be allocated depending on the value of the tag. Consider the following example, using Pascal's variant record notation.

```
shape = record
```



```

    area : real;
    case kind : shapeKind of
        square : (side : real);
        circle : (radius : real)
    end;
end;

```

This definition creates a variant record named `shape`. The field `area` is always present (since all shapes have an area). Two kinds of shapes are possible—squares and circles. If `kind` is set to `square`, field `side` is allocated. If `kind` is set to `circle`, field `radius` is allocated. Otherwise, no additional fields are allocated.

Since fields in different variants are mutually exclusive, we don't allocate space for them as we do for ordinary fields. Rather, the variants **overlay** each other. In our example, `side` and `radius` share the same location in the `shape` record, since only one is "active" in any record.

When we allocate offsets for fields in a variant record, we start the first field of each variant at the same offset, just past the space allocated for the tag. For `shape` we would have

Field	Size	Offset
<code>area</code>	4	0
<code>kind</code>	1	4
<code>side</code>	4	8
<code>radius</code>	4	8

The size assigned to `shape` (with proper alignment) is 12 bytes—enough space for `area`, `kind` and either `side` or `radius`.

As with ordinary structures and records, we address fields by adding their offset to the record's starting address. Fields within a variant are accessible only if the tag is properly set. This can be checked by first comparing the tag field with its expected value before allowing access to a variant field. Thus if we wished to access field `radius`, we would first check that `kind` contained the bit pattern assigned to `circle`. (Since this check can slow execution, it is often disabled, just as array bound checking is often disabled.)

Java contains no variant records, but achieves the same capabilities by using subclasses. A class representing all the fields except the tag and variants is first created. Then a number of subclasses are created, one for each desired variant. No explicit tag is needed—the name assigned to each subclass acts like a tag. For example, for shapes we would define.

```

class Shape {
    float area;
}
class Square extends Shape {
    float side;
}

```

```
class Circle extends Shape {
    float radius;
}
```

Ordinary JVM bytecodes for fields (`getfield` and `putfield`) provide access and security. Before a field is accessed, its legality (existence and type) are checked. Hence it is impossible to access a field in the “wrong” subclass. If necessary, we can check which subclass (that is, which kind of shape) we have using the `instanceof` operator.

Pascal also allows variant records without tags; these are equivalent to the union types found in C and C++. In these constructs fields overlay each other, with each field having an offset of 0. As one might expect, such constructs are quite error-prone, since it is often unclear which field is the one to use. In the interests of security Java does not provide a union type.

14.4 Arrays

14.4.1 Static One-Dimensional Arrays

One of the most fundamental data structures in programming languages is the array. The simplest kind of array is one that has a single index and static (constant) bounds. An example (in C or C++) is

```
int a[100];
```

Essentially an array is allocated as a block of N identical data objects, where N is determined by the declared size of the array. Hence in the above example 100 consecutive integers are allocated.

An array, like all other data structures, has a size and an alignment requirement. An array’s size is easily computed as

$$\text{size}(\text{array}) = \text{NumberOfElements} * \text{size}(\text{Element}).$$

If the bounds of an array are included within it (as is the case for Java), the array’s size must be increased accordingly.

An array’s alignment restriction is that of its components. Thus an integer array must be word-aligned if integers must be word-aligned.

Sometimes padding is used to guarantee alignment of **all** elements. For example, given the C declaration

```
struct s {int a; char b;} ar[100];
```

each element (a struct) is padded to a size of 8 bytes. This is necessary to guarantee that `ar[i].a` is always word-aligned.

When arrays are assigned, size information is used to determine how many bytes to copy. A series of load store instructions can be used, or a copy loop, depending on the size of the array.

In C, C++ and Java, all arrays are **zero-based** (the first element of an array is always at position 0). This rule leads to a very simple formula for the address of an array element:

$$\text{address}(A[i]) = \text{address}(A) + i * \text{size}(\text{Element})$$

For example, using the declaration of `ar` as an array of struct `s` given above,
 $\text{address}(\text{ar}[5]) = \text{address}(\text{ar}) + 5 * \text{size}(\text{s}) = \text{address}(\text{ar}) + 5 * 8 = \text{address}(\text{ar}) + 40$.
 Computing the address of a field within an array of structures is easy too. Recall that

$\text{address}(\text{struct}.\text{field}) = \text{address}(\text{struct}) + \text{offset}(\text{field})$.

Thus $\text{address}(\text{struct}[i].\text{field}) =$

$\text{address}(\text{struct}[i]) + \text{offset}(\text{field}) = \text{address}(\text{struct}) + i * \text{size}(\text{struct}) + \text{offset}(\text{field})$.

For example, $\text{address}(\text{ar}[5].\text{b}) =$

$\text{address}(\text{ar}[5]) + \text{offset}(\text{b}) = \text{address}(\text{ar}) + 40 + 4 = \text{address}(\text{ar}) + 44$.

In Java, arrays are allocated as objects; all the elements of the array are allocated within the object. Exactly how the objects are allocated is unspecified by Java's definition, but a sequential contiguous allocation, just like C and C++, is the most natural and efficient implementation.

Java hides explicit addresses within the JVM. Hence to index an array it is not necessary to compute the address of a particular element. Instead, special array indexing instructions are provided. First, an array object is created and assigned to a field or local variable. For example,

```
int ar[] = new int[100];
```

In Java, an array assignment just copies a reference to an array object; no array values are duplicated. To create a copy of an array, the `clone` method is used. Thus `ar1 = ar2.clone()` gives `ar1` a newly created array object, initially identical to `ar2`.

To load an array element (onto the stack top), we first push a reference to the array object and the value of the index. Then an "array load" instruction is executed. There is actually a family of array load instructions, depending on the type of the array element being loaded: `iaload` (integer array element), `faload` (floating array element), `daload` (double array element), etc. The first letter of the opcode denotes the element type, and the suffix "aload" denotes an array load.

For example, to load the value of `ar[5]`, assuming `ar` is a local array given a frame index of 3, we would generate

```
aload 3 ; Load reference to array ar
iconst_5 ; Push 5 onto stack
iaload ; Push value of ar[5] onto the stack
```

Storing into an array element in Java is similar. We first push three values: a reference to an array object, the array index, and the value to be stored. The array store instruction is of the form `Xastore`, where `X` is one of `i`, `l`, `f`, `d`, `a`, `b`, `c`, or `s` (depending on the type to be stored). Hence to implement `ar[4] = 0` we would generate

```
load 3 ; Load reference to array ar
iconst_4 ; Push 4 onto stack
iconst_0 ; Push 0 onto stack
iastore ; Store 0 into ar[4]
```

Array Bounds Checking. An array reference is legal only if the index used is in bounds. References outside the bounds of an array are undefined and dangerous,

as data unrelated to the array may be read or written. Java, with its attention to security, checks that an array index is in range when an array load or array store instruction is executed. An illegal index forces an `ArrayIndexOutOfBoundsException`. Since the size of an array object is stored within the object, checking the validity of an index is easy, though it does slow access to arrays.

In C and C++ indices out of bounds are also illegal. Most compilers do not implement bounds checking, and hence program errors involving access beyond array bounds are common.

Why is bounds checking so often ignored? Certainly speed is an issue. Checking an index involves two checks (lower and upper bounds) and each check involves several instructions (to load a bound, compare it with the index, and conditionally branch to an error routine). Using unsigned arithmetic, bounds checking can be reduced to a single comparison (since a negative index, considered unsigned, looks like a *very* large value). Using the techniques of Chapter 16, redundant bounds checks can often be optimized away. Still, array indexing is a very common operation, and bounds checking adds a real cost (though buggy programs are costly too!).

A less obvious impediment to bounds checking in C and C++ is the fact that array names are often treated as equivalent to a pointer to the array's first element. That is, an `int []` and a `*int` are often considered the same. When an array pointer is used to index an array, we don't know what the upper bound of the array is. Moreover, many C and C++ programs intentionally violate array bounds rules, initializing a pointer one position before the start of an array or allowing a pointer to progress one position past the end of an array.

We can support array bounds checking by including a "size" parameter with every array passed as a parameter and every pointer that steps through an array. This size value serves as an upper bound, indicating the extent to access allowed. Nevertheless, it is clear that bounds checking is a difficult issue in languages where difference between pointers and array addresses is blurred.

Array parameters often require information beyond a pointer to the array's data values. This includes information on the array's size (to implement assignment) and information on the array's bounds (to allow subscript checking). An array descriptor (sometimes called a **dope vector**), containing this information, can be passed for array parameters instead of just a data pointer.

Non-zero Lower Bounds. In C, C++ and Java, arrays always have a lower bound of zero. This simplifies array indexing. Still, a single fixed lower bound can lead to clumsy code sequences. Consider an array indexed by years. Having learned not to represent years as just two digits, we'd prefer to use a full four digit year as an index. Assuming we really want to use years of the twentieth and twenty-first centuries, an array starting at 0 is very clumsy. So is explicitly subtracting 1900 from each index before using it.

Other languages, like Pascal and Ada, have already solved this problem. An array of the form `A [low . . high]` may be declared, where all indices in the range

`low`, ..., `high` are allowed. With this array form, we can easily declare an array index by four digit years: `data[1900..2010]`.

With a non-zero lower bound, our formula for the size of an array must be generalized a bit:

$$\text{size(array)} = (\text{UpperBound} - \text{LowerBound} + 1) * \text{size(Element)}.$$

How much does this generalization complicate array indexing? Actually, surprisingly little. If we take the Java approach, we just include the lower bound as part of the array object we allocate. If we compute an element address in the code we generate, the address formula introduced above needs to be changed a little:

$$\text{address}(A[i]) = \text{address}(A) + (i - \text{low}) * \text{size}(\text{Element})$$

We subtract the array's lower bound (`low`) before we multiply by the element size. Now it is clear why a lower bound of zero simplifies indexing—a subtraction of zero from the array index can be skipped. But the above formula can be rearranged to:

$$\begin{aligned} \text{address}(A[i]) &= \text{address}(A) + (i * \text{size}(\text{Element})) - (\text{low} * \text{size}(\text{Element})) = \\ &= \text{address}(A) - (\text{low} * \text{size}(\text{Element})) + (i * \text{size}(\text{Element})) \end{aligned}$$

We now note that `low` and `size(Element)` are normally compile-time constants, so the expression `(low * size(Element))` can be reduced to a single value, `bias`. So now we have

$$\text{address}(A[i]) = \text{address}(A) - \text{bias} + (i * \text{size}(\text{Element}))$$

The address of an array is normally a static address (a global) or an offset relative to a frame pointer (a local). In either case, the `bias` value can be folded into the array's address, forming a new address or frame offset reduced by the value of `bias`.

For example, if we declare an array `int data[1900..2010]`, and assign `data` an address of 10000, we get a `bias` value of `1900*size(int) = 7600`. In computing the address of `data[i]` we compute `2400+i*4`. This is exactly the same form that we used for zero-based arrays.

Even if we allocate arrays in the heap (using `new` or `malloc`), we can use the same approach. Rather than storing a pointer to the array's first element, we store a pointer to its zero-th element (which is what subtracting `bias` from the array address gives us). We do need to be careful when we assign such arrays though; we must copy data from the first valid position in the array. Still, indexing is far more common than copying, so this approach is still a good one.

Dynamic and Flex Arrays. Some languages, including Algol 60, Ada and Java support **dynamic arrays** whose bounds and size are determined at run-time. When the scope of a dynamic array is entered, its bounds and size are evaluated and fixed. Space for the array is then allocated. The bounds of a dynamic array may include parameters, variables and expressions. For example, in C extended to allow dynamic arrays, procedure `P` we might include the declaration

```
int examScore[numOfStudents()];
```

Because the size of a dynamic array isn't known at compile-time, we can't allocate space for it globally or in a frame. Instead, we must allocate space either on the stack (just past the current frame) or in the heap (Java does this). A pointer

to the location of the array is stored in the scope in which the array is declared. The size of the array (and perhaps its bounds) are also stored. Using our above example, we would allocate space for `examScores` as shown in Figure 14.1. Within `P`'s frame we allocate space for a pointer to `examScore`'s values as well as its size.

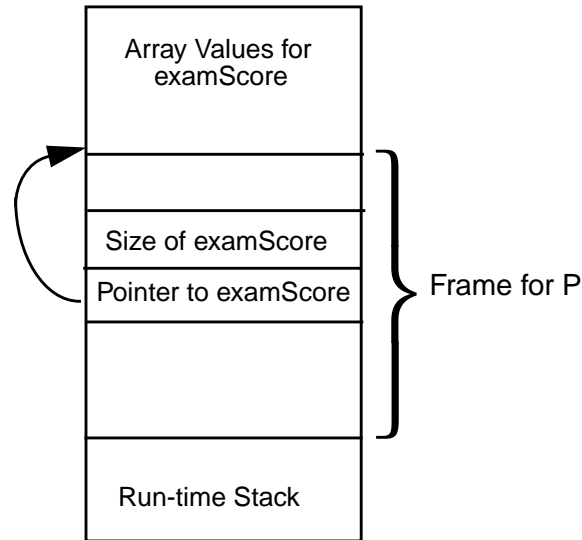


Figure 14.1 Allocation of a Dynamic Array

Accessing a dynamic array requires an extra step. First the location of the array is loaded from a global address or from an offset within a frame. Then, as usual, an offset within the array is computed and added to the array's starting location.

A variant of the dynamic array is the **flex array**, which can expand its bounds during execution. (The Java `Vector` class implements a flex array.) When a flex array is created, it is given a default size. If during execution an index beyond the array's current size is used, the array is expanded to make the index legal. Since the ultimate size of a flex array isn't known when the array is initially allocated, we store the flex array in the heap, and point to it. Whenever a flex array is indexed, the array's current size is checked. If it is too small, another larger array is allocated, values from the old allocation are copied to the new allocation, and the array's pointer is reset to point to the new allocation.

When a dynamic or flex array is passed as a parameter, it is necessary to pass an array descriptor that includes a pointer to the array's data values as well as information on the array's bounds and size. This information is needed to support indexing and assignment.

14.4.2 Multidimensional Arrays

In most programming languages multidimensional arrays may be treated as arrays of arrays. In Java, for example, the declaration

```
int matrix[] [] = new int [5] [10];
```

first assigns to `matrix` an array object containing five references to integer arrays. Then, in sequence, five integer arrays (each of size ten) are created, and assigned to the array `matrix` references (Figure 14.2).

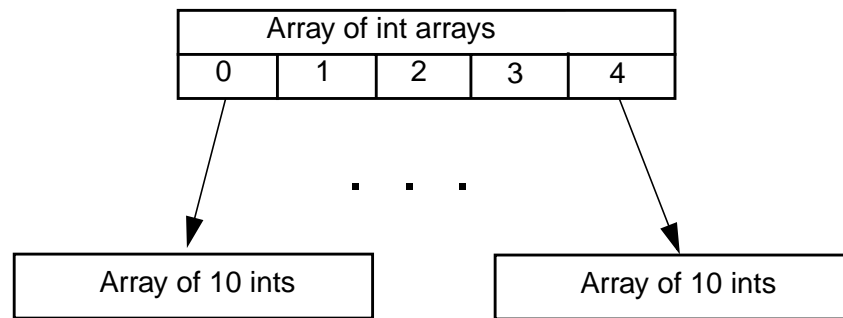


Figure 14.2 A Multidimensional Array in Java

Other languages, like C and C++, allocate one block of memory, sufficient to contain all the elements of the array. The array is arranged in **row-major** form, with values in each row contiguous and individual rows placed sequentially (Figure 14.3). In row-major form, multidimensional arrays really are arrays of arrays, since in an array reference like `A[i][j]`, the first index (`i`) selects the `i`-th row, and the second index (`j`) chooses an element within the selected row.

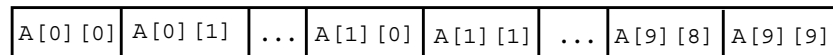


Figure 14.3 Array `A[10][10]` Allocated in Row-Major Order

An alternative to row-major allocation is **column-major** allocation, which is used in Fortran and related languages. In column-major order values in individual columns are contiguous, and columns are placed adjacent to each other (Figure 14.4). Again, the whole array is allocated as a single block of memory.

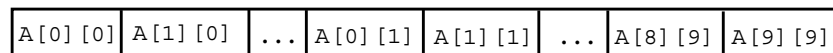


Figure 14.4 Array `A[10][10]` Allocated in Column-Major Order

How are elements of multidimensional arrays accessed? For arrays allocated in row-major order (the most common allocation choice), we can exploit the fact that multidimensional arrays can be treated as arrays of arrays. In particular, to compute the address of $A[i][j]$, we first compute the address of $A[i]$, treating A as a one dimensional array of values that happen to be arrays. Once we have the address of $A[i]$, we then compute the address of $X[j]$, where X is the starting address of $A[i]$.

Let's look at the actual computations needed. Assume array A is declared to be an n by m array (e.g., it is declared as $T A[n][m]$, where T is the type of the array's elements).

We now know that

$$\text{address}(A[i][j]) = \text{address}(X[j]) \text{ where } X = \text{address}(A[i]).$$

$$\text{address}(A[i]) = \text{address}(A) + i * \text{size}(T) * m.$$

Now

$$\text{address}(X[j]) = \text{address}(X) + j * \text{size}(T).$$

Putting these together,

$$\text{address}(A[i][j]) = \text{address}(A) + i * \text{size}(T) * m + j * \text{size}(T) =$$

$$\text{address}(A) + (i * m + j) * \text{size}(T).$$

Computation of the address of elements in a column-major array is a bit more involved, but we can make a useful observation. First, recall that **transposing** an array involves interchanging its rows and columns. That is, the first column of the array becomes the first row of the transposed array, the second column becomes the second row and so on (see Figure 14.5).

1	6
2	7
3	8
4	9
5	10

Original Array

1	2	3	4	5
6	7	8	9	10

Transposed Array

Figure 14.5 An Example of Array Transposition

Now observe that a column-major ordering of elements in an array corresponds to a row-major ordering in the transposition of that array. Allocate an n by m array, A , in column-major order and consider any element $A[i][j]$. Now transpose A into AT , an m by n array, and allocate it in row-major order. Element $AT[j][i]$ will always be the same as $A[i][j]$.

What this means is that we have a clever way of computing the address of $A[i][j]$ in a column-major array. We simply compute the address of $AT[j][i]$, where AT is treated as a row-major array with the same address as A , but with interchanged row and column sizes ($A[n][m]$ becomes $AT[m][n]$).

As an example, refer to Figure 14.5. The array on the left represents a 5 by 2 array of integers; in column-major order, the array's elements appear in the order 1 to 10. Similarly, the array on the right represents a 2 by 5 array of integers; in row-major order, the array's elements appear in the order 1 to 10. It is easy to see that a value in position $[i][j]$ in the left array always corresponds to the value at $[j][i]$ in the right (transposed) array.

In Java, indexing multidimensional arrays is almost exactly the same as indexing one dimensional arrays. The same JVM bytecodes are used. The only difference is that now more than one index value must be pushed. For example, if A is a two dimensional integer array (at frame offset 3), to obtain the value of $A[1][2]$ we would generate

```

aload 3 ; Load reference to array A
iconst_1 ; Push 1 onto stack
iconst_2 ; Push 2 onto stack
iaload ; Push value of A[1][2] onto the stack

```

The JVM automatically checks that the right number of index values are pushed, and that each is in range.

14.5 Implementing Other Types

In this section we'll consider implementation issues for a number of other types that appear in programming languages.

Enumerations. A number of programming languages, including C, C++, Pascal and Ada, allow users to define a new data type by enumerating its values. For example, in C we might have

```
enum color { red, blue, green } myColor;
```

An enumeration declaration involves both semantics and code generation. A new type is declared (e.g. `color`). Constants of the new type are also declared (`red`, `blue` and `green`). Finally, a variable of the new type (`myColor`) is declared.

To implement an enumeration type we first have to make values of the enumeration available at run-time. We assign internal encoding to the enumeration's constant values. Typically these are integer values, assigned in order, starting at zero or one. Thus we might treat `red` as the value 1, `blue` as the value 2 and `green` as the value 3. (Leaving zero unassigned makes it easy to distinguish uninitialized variables, which often are set to all zeroes). In C and C++, value assignment starts at zero.

C and C++ allow users to set the internal representation for selected enumeration values; compiler selected values must not interfere with those chosen by a programmer. We can allocate initialized memory locations corresponding to each

enumeration value, or we can store the encoding in the symbol table and fill them directly into instructions as immediate operands.

Once all the enumeration values are declared, the size required for the enumeration is chosen. This is usually the smallest “natural” memory unit that can accommodate all the possible enumeration values. Thus we might choose to allocate a byte for variables of type `color`. We could actually use as few as two bits, but the cost of additional instructions to insert or extract just two bits from memory makes byte-level allocation more reasonable.

A variable declared to be an enumeration is implemented just like other scalar variables. Memory is allocated using the size computed for the enumeration type. Values are manipulated using the encoding chosen for the enumeration values.

Thus variable `myColor` is implemented as if it were declared to be of type `byte` (or `char`). The assignment `myColor = blue` is implemented by storing 2 (the encoding of `blue`) into the byte location assigned to `myColor`.

Subranges. Pascal and Ada include a useful variant of enumerated and integer types—the subrange. A selected range of values from an existing type may be chosen to form a new type. For example

```
type date = 1..31;
```

A subrange declaration allows a more precise definition of the range of values a type will include. It also allows smaller memory allocations if only a limited range of values is possible.

To implement subranges, we first decide on the memory size required. This is normally the smallest “natural” unit of allocation sufficient to store all the possible values. Thus a byte would be appropriate for variables of type `date`. For subranges involving large values, larger memory sizes may be needed, even if the range of values is limited. Given the declaration

```
type year = 1900..2010;
```

we might well allocate a half word, even though only 111 distinct values are possible. Doing this makes loading and storing values simpler, even if data sizes are larger than absolutely necessary.

In manipulating subrange values, we may want to enforce the subrange’s declared bounds. Thus whenever a value of type `year` is stored, we can generate checks to verify that the value is in the range 1900 to 2010. Note that these checks are very similar to those used to enforce array bounds.

Ada allows the bounds of a subrange to be expressions involving parameters and variables. When range bounds aren’t known in advance, a maximum size memory allocation is made (full or double word). A run-time descriptor is created and initialized to the range bounds, once they are evaluated. This descriptor is used to implement run-time range checking.

Files. Most programming languages provide one or more file types. File types are potentially complex to implement since they must interact with the computer’s operating system. For the most part, a compiler must allocate a “file descriptor”

for each active file. The format of the file descriptor is system-specific; it is determined by the rules and conventions of the operating system in use.

The compiler must fill in selected fields from the file's declaration (name of the file, its access mode, the size of data elements, etc.). Calls to system routines to open and close files may be needed when the scope containing a file is entered or exited. Otherwise, all the standard file manipulation operations—read, write, test for end-of-file, etc., simply translate to calls to corresponding system utilities.

In Java, where all files are simply instances of predefined classes, file manipulation is no different than access to any other data object.

Pointers and Objects. We discussed the implementation of pointers and object references in Section 12.7. In Java, all data except scalars are implemented as objects and accessed through object references (pointers). Other languages routinely use a similar approach for complex data objects. Hence, as we learned above, dynamic and flex arrays are allocated on the stack or heap, and accessed (invisibly) through pointers. Similarly, strings are often implemented by allocating the text of the string in the heap, and keeping a pointer to the current text value in the heap.

In Java, all objects are referenced through pointers, so the difference between a pointer and the object it references is obscured. In languages like C and C++ that have both explicit and implicit pointers, the difference between pointer and object can be crucial.

A good example of this is the difference between array and structure assignment in C and C++. The assignment `struct1 = struct2` is implemented as a memory copy; all the fields within `struct2` are copied into `struct1`. However, the assignment `array1 = array2` means something very different. `array2` represents the *address* of `array2`, and assigning it to `array1` (a constant address) is illegal.

Whenever we blur the difference between a pointer and the object it references, similar ambiguities arise. Does `ptr1 = ptr2` mean copy a pointer or copy the object pointed to? In implementing languages it is important to be clear on exactly when evaluation yields a pointer and when it yields an object.

Exercises

1. Local variables are normally allocated within a frame, providing for automatic allocation and deallocation when a frame is pushed and popped. Under what circumstance must a local variable be dynamically allocated? Are there any advantages to allocating a local variable statically (i.e., giving it a single fixed address)? Under what circumstances is static allocation for a local permissible?

2. Consider the following C/C++ structure declarations


```
struct {int a; float b; int c[10];} s;
struct {int a; float b; } t[10];
```

Choose your favorite computer architecture. Show the code that would be generated for `s.c[5]` assuming `s` is statically allocated at address 1000. What code would be generated for `t[3].b` if `t` is allocated within a frame at offset 200?

3. Consider the following Java class declaration


```
class C1 {int a; float b; int c[]; C1 d[]};
```

Assume that `v` is a local reference to `C1`, with an index of 4 within its frame. Show the JVM code that would be generated for `v.a`, `v.c[5]`, `v.d[2].b` and `v.d[2].c[4]`.

4. Explain how the following three declaration would be translated in C, assuming that they are globally declared (outside all subprogram bodies).


```
const int ten = 10;
const float pi = 3.14159;
const int limit = get_limit();
```

How would the above three declaration be translated if they are locals declarations (within a subprogram body)?

5. Assume that in C we have the declaration `int a[5][10][20]`, where `a` is allocated at address 1000. What is the address of `a[i][j][k]` assuming `a` is allocated in row-major order? What is the address of `a[i][j][k]` assuming `a` is allocated in column-major order?
6. Most programming languages (including Pascal, Ada, C, and C++) allocate global aggregates (records, arrays, structs and classes) statically, while local aggregates are allocated within a frame. Java, on the other hand, allocates all aggregates in the heap. Access to them is via object references allocated statically or within a frame. Is it less efficient to access an aggregate in Java because of its mandatory heap allocation? Are there any advantages to forcing all aggregates to be uniformly allocated in the heap?
7. In Java, subscript validity checking is mandatory. Explain what changes would be needed in C or C++ (your choice) to implement subscript validity checking. Be sure to address the fact that pointers are routinely used to access array elements. Thus you should be able to checks array accesses that are done

through pointers, including pointers that have been incremented or decremented.

8. Assume we add a new option to C++ array that are heap-allocated, the **flex** option. A flex array is automatically expanded in size if an index beyond the array's current upper limit is accessed. Thus we might see

```
ar = new flex int [10]; ar[20] = 10;
```

The assignment to position 20 in `ar` forces an expansion of `ar`'s heap allocation. Explain what changes would be needed in array accessing to implement flex arrays. What should happen if an array position beyond an array's current upper limit is read rather than written?

9. Fortran library subprograms are often called from other programming languages. Fortran assumes that multi-dimensional arrays are stored in column-major order; most other languages assume row-major order. What must be done if a C program (which uses row-major order) passes a multi-dimensional array to a Fortran subprogram. What if a Java method, which stores multi-dimensional arrays as arrays of array object references, passes such an array to a Fortran subprogram?
10. Recall that offsets within a record or struct must sometimes be adjusted upward due to alignment restrictions. Thus in the following two C structs, `S1` requires 6 bytes whereas `S2` requires only 4 bytes.

```
struct {          struct {
    char  c1;      char  c1;
    short s;      char  c2;
    char  c2;      short s;
} S1;            } S2;
```

Assume we have a list of the fields in a record or struct. Each is characterized by its size and alignment restriction 2^a . (A field with an alignment restriction 2^a must be assigned an offset that is a multiple of 2^a).

Give an algorithm that determines an ordering of fields that minimizes the overall size of a record or struct while maintaining all alignment restrictions. How does the execution time of your algorithm (as measured in number of execution steps) grow as the number of fields increases?

15

Code Generation

In this chapter we will explore how intermediate forms, like JVM bytecodes, are translated into executable form. Collectively, this process is called *code generation*, though code generation actually involves a number of individual tasks that must be handled.

Translation has already been done by code generation subroutines associated with AST nodes (Chapters 12 to 14). Source level constructs have been transformed in bytecodes. Bytecodes may be interpreted by a byte code interpreter. Alternatively, we may wish to complete the translation process and produce machine instructions native to the computer we are using.

The first problem we will face is *instruction selection*. Instruction selection is highly target machine dependent; it involves choosing a particular instruction sequence to implement a given JVM bytecode. Even for a simple bytecode we may have a choice of possible implementations. For example, an `inc` instruction, that adds a constant to a local variable, might be implemented by loading a variable into a register, loading the constant into a second register, doing a register to register add, and storing the result back into the variable. Alternatively, we might choose to keep the variable in a register, allowing implementation using only a single add immediate instruction.

Besides instruction selection, we must also deal with *register allocation and code scheduling*. Register allocation aims to use registers effectively, by minimizing *register spilling* (storing a value held in a register and reloading the register with

something else). Even a few unnecessary loads and stores can significantly reduce the speed of an instruction sequence. *Code scheduling* worries about the order in which generated instructions are executed. Not all valid instruction ordering are equally good—some incur unnecessary delays that must be avoided.

We shall first consider how bytecodes may be efficiently translated to machine-level instructions. Techniques that optimize the code we generate will be considered next, especially the translation of expressions in tree forms. A variety of techniques that support efficient use of registers, especially *graph coloring* will be discussed. Approaches to code scheduling will next be studied. Techniques that allow us to easily and automatically retarget a code generator to a new computer will then be discussed. Finally, a form of optimization that is particularly useful at the code generation level—peephole optimization—will be studied.

15.1 Translating Bytecodes

We will first consider how to translate the bytecodes generated by the translation phase of our compiler into conventional machine code. Many different machine instruction sets exist; one for each computer architecture. Examples include the Intel x86 architecture, the SPARC, the Alpha, the Power PC, and the MIPS.

In this chapter we'll use the MIPS R3000 instruction set. This architecture is clean and easy to use and yet represents well the current generation of RISC architectures. The MIPS R3000 also is supported by SPIM [Lar 93] a widely-available MIPS interpreter written in C.

Most bytecodes map directly into one or two MIPS instructions. Thus an `iadd` instruction corresponds directly to the MIPS `add` instruction. The biggest difference in the design of bytecodes and the MIPS (or any other modern architecture) is that bytecodes are stack-oriented whereas the MIPS is register-oriented.

The most obvious approach to handling stack based operands is to load top of stack values into registers when they are used, and to push registers onto the stack when values are computed. This unfortunately is also one of the *worst* approaches. The problem is that explicit pop and push operations imply memory load and store instructions which can be slow and bulky.

Instead, we'll make a few simple, but important, observations on how stack operands are used. First, note that no operands are left on the stack between source-level statements. If they were, a statement, placed in a loop, could cause stack overflow. Thus the stack is used only to hold operands while parts of a statement are executed. Moreover, each stack operand is "touched" twice—when it is created (pushed) and when it is used (popped).

These observations allow us to map stack operands directly into registers—no pushes or pops of the run-time stack are really needed. We can imagine the JVM operand stack as containing register names rather than values. When a particular value is at the top of the stack, we'll use the corresponding "top register" as the source of our operand. This may seem complex, but it really is quite simple. Consider the Java assignment statement `a = b + c - d;` (where `a`, `b`, `c` and `d` are integer locals). The corresponding bytecodes are


```

iload 2 ; Push int b onto stack
iload 3 ; Push int c onto stack
iadd    ; Add top two stack values
iload 4 ; Push int d onto stack
isub    ; Subtract top two stack values
istore 1 ; Store top stack value into a

```

Whenever a value is pushed, we will create a temporary location to hold it. This temporary location (usually just called a temporary) will normally be allocated to a register. We'll track the names of the temporaries associated with stack locations as bytecodes are processed. At any point, we'll know exactly what temporaries are logically on the stack. We say logically, because these values aren't pushed and popped at run time. Rather, values are directly accessed from the registers that hold them.

Continuing with our example, assume *a*, *b*, *c* and *d* are assigned frame offsets 12, 16, 20 and 24 respectively (we'll discuss memory allocation for locals and fields below). These four variables are given offsets because, as discussed in Chapter 11, local variables in a procedure or method are allocated as part of a frame—a block of memory allocated (on the run-time stack) whenever a call is made. Thus rather than push or pop individual data values, as bytecodes do, we prefer to push a single large block of memory once per call.

Let's assume the temporaries we allocate are MIPS registers, denoted *\$t0*, *\$t1*, Each time we generate code for a bytecode that pushes a value onto the stack, we'll call `getReg` (Section 15.3.1) to allocate a result register. Whenever we generate MIPS code for a bytecode that accesses stack values, we'll use the registers already allocated to hold stack values. The net effect is to use registers rather than stack locations to hold operands, which yields fast and compact instruction sequences. For our above example we might generate

```

lw    $t0,16($fp) # Load b, at 16+$fp, into $t0
lw    $t1,20($fp) # Load c, at 20+$fp, into $t1
add   $t2,$t0,$t1 # Add $t0 and $t1 into $t2
lw    $t3,24($fp) # Load d, at 24+$fp, into $t3
sub   $t4,$t2,$t3 # Subtract $t3 from $t2 into $t4
sw    $t4,12($fp) # Store result into a, at 12+$fp

```

The `lw` instruction loads a word of memory into a register. Addresses of locals are computed relative to `$fp`, the frame pointer, which always points to the currently active frame. Similarly, `sw` stores a register into a word of memory. `add` and `sub` add (or subtract) two registers, putting the result into a third register.

Bytecodes that push constants, like `bipush n`, can be implemented as a load immediate of a literal into a register. As an optimization, we can delay doing the actual load until we're sure it is needed. That is, we note that a stack location, associated with a MIPS register, will hold a known literal value. When that register is used, we may be able to replace the register with the value it must hold (for example, replacing a register to register `add`, with an `add immediate` instruction).

Allocating memory addresses. As we learned in Chapter 11, local variables and parameters are allocated in the frame associated with a procedure or method. Thus we must map each JVM local variable into a unique frame offset, used to address the variable in load and store instructions. Since a frame contains some fixed-size control information followed by local data, a simple formula like $\text{offset} = \text{const} + \text{size} * \text{index}$ suffices, where *index* is the JVM index assigned to a variable, *size* is the size (in bytes) of each stack value, *const* is the size (in bytes) of the fixed-size control area in the frame, and *offset* is the frame offset used in generated MIPS code.

Static fields of classes are assigned fixed, static addresses when a class is compiled. These addresses are used whenever a static field is referenced. Instance fields are accessed as an offset relative to the beginning of a class. Thus if we had a class `Complex` defined as

```
class Complex { float re; float im; }
```

the two fields `re` and `im`, each one word in size, would be given offsets of 0 (for `re`) and 4 (for `im`) within instances of the class. The bytecode `getField Complex/im` fetches field `im` of the `Complex` object referenced by the top of stack. Translation is easy. We first lookup the offset of field `im` in class `Complex`, which is 4. The reference to the referenced object is in the register corresponding to top-of-stack, say `$t0`. We could add 4 to `$t0`, but since the MIPS has an indexed addressing mode that adds a constant to a register automatically (denoted `const($reg)`), we need generate no code. We simply generate `lw $t1,4($t0)`, which loads the field into register `$t1`, which now corresponds to the top of stack.

Allocating Arrays and Objects. In Java, and hence in the JVM, all objects are allocated on the heap. To translate a `new` or `newarray` bytecode, we'll need to call a heap-allocation subroutine, like `malloc` in C. We pass the size of the object required and receive a pointer to a newly allocated block of heap memory. For a `new` bytecode the size required is determined by the number and size of the fields in the object. In addition, a fixed-size header (to store the size and type of the object) is required. The overall memory size needed can be computed when the class definition of the object is compiled. In our earlier example of a `Complex` object, the size required would be 8 bytes plus 2 or 4 bytes of header information.

For `newarray` we determine the size required by multiplying the number of elements requested (in the register corresponding to the top of stack) by size requirement for individual array elements (stored in the symbol table entry for the requested class). Again, space for a fixed-size object header must be included.

Default initialization of fields within objects must also be performed by clearing or copying appropriate bit patterns (based on type declarations).

In languages like C and C++, objects and arrays can be allocated “inline” in the current frame if their size is known at compile-time. We can do a similar allocation within the current frame if we know that no reference to the allocated object “escapes.” That is, if no reference to the allocated object or array is assigned to a field or returned as a function value, then the object or array is no


```

add    $temp,$temp,$array # Compute $array + 4*$index
sw     $val,OFFSET($temp) # Load $val into word at
                                # $array + 4*$index + OFFSET

```

The MIPS code we’ve chosen for array indexing looks rather complex and expensive, especially since arrays are a very commonly used data structure. Part of this complexity is due to the fact that we’ve included array bounds checking, as required in Java. In C and C++, array bounds are rarely checked at run-time, allowing for fast (and less secure!) code.

In many cases it is possible to optimize or entirely eliminate array bounds checks. On architectures that support unsigned arithmetic, the check for an index too large and the check for an index too small (less than zero) can be combined! The trick is to do an unsigned compare between the array index and the array size. A negative index will be equivalent to a *very* large unsigned value (since its left-most bit will be one), making it greater than the array size.

In a for loop it is often possible to determine that a loop index is bounded by known lower and upper bounds. With this information, arrays indexed by the loop index may be known to be “in range,” eliminating any need for explicit checking. Similarly, once an array bound is checked, subsequent checks of the same bound are unnecessary until the index is changed. Thus in `a[i] = 100 - a[i]`, `a[i]` needs to be checked only once.

If array bounds checks are optimized away, or simply suppressed, array indexing is much more efficient—typically three (or fewer) instructions (a shift (or multiply), an add, and a load or store). In the case that the array index is a compile-time constant (e.g., `a[100]`), we can reduce this to a single instruction by doing the computation of `size*index+offset` at compile-time and using it directly in a load or store instruction.

Method Calls. The JVM makes method calls very simple. Many of the details of a call are hidden. In implementing an `invokestatic` or `invokevirtual` bytecode, we must make such hidden details explicit.

Let’s look at `invokestatic` first. In the bytecode version of the call, parameters are pushed onto the stack, and a static method, qualified by its class and type (to support overloading) is accessed. In our MIPS translation, the parameters will be in registers, which is fine since that’s how most current architectures pass scalar (word-sized) parameters.

We’ll need to guarantee that parameters are placed in the right registers. On the MIPS, the first four scalar parameters are passed in registers `$a0` to `$a3`; remaining parameters and non-scalar parameters are pushed onto the run-time stack. In our translation, we can generate explicit register copy instructions to move argument values (already in registers) to the correct registers. If we wish to avoid these extra copy instructions, we can compute the parameters directly into the correct registers. This is called register targeting. Essentially, when the parameter is computed, we mark the target register into which the parameter will be computed to be the appropriate argument register. Graph coloring, as discussed in Section 15.3.2, makes targeting fairly easy to do. For parameters that are to be

passed on the run-time stack, we simply extend the stack (by adjusting the top-of-stack register, `$sp`) and then store excess parameters into the locations just added.

To transfer control to the subprogram, we issue a `jal` (jump and link) instruction. This instruction transfers control to the start of the method to be called (using an address recorded when the subprogram was translated) *and* stores a return address in the return address register, `$ra`.

Additional details must be handled to complete our translation of an `invokestatic` instruction. Since at the point of call variable and expression values may be held in registers, these registers must be saved prior to execution of the method. All registers that hold values that may be destroyed during the call (by the instructions in the method body) are saved on the stack, and restored after the method completes execution. Registers may be saved by the caller (these are caller-save registers) or by the method to be called (these are callee-save registers). It doesn't really matter if the caller or callee does the saving (often both save selected registers), but any register holding a program value needed after the call must be protected.

If a non-local or global variable is held in a register, it must be saved, prior to a call, in its assigned memory location. This guarantees that the subprogram will see the correct value during the call. Upon return, registers holding non-local or global variables must be reloaded since the subprogram may have updated their values.

As an example, consider this function call `a = f(i, 2)`; `a` is a static field, `f` is a static method and `i` is a local variable held in register `$t0`, a caller-save register. Assume that a storage temporary, assigned frame offset 32, is created to hold the value of `$t0` across the call. The following MIPS code is produced

```

move  $a0,$t0           # Copy $t0 to parm register 1
li    $a1,2             # Load 2 into parm register 2
sw    $t0,32($fp)       # Store $t0 across call
jal   f                 # Call function f
                                # Function value is in $v0
lw    $t0,32($fp)       # Restore $t0
sw    $v0,a             # Store function value in a

```

When a method is called, space for its frame must be pushed, and the frame and stack pointers must be properly updated. This is normally done in the prologue of the called method, just before its body is executed. Similarly, after a method is finished, its frame must be popped, and frame and stack pointers properly reset. This is done in the method's epilogue, just before branching back to the caller's return address. The exact code sequences needed vary according to hardware, and operating system conventions, but the following MIPS instructions can be used to push, and later pop, a method's frame. (The size of the frame, `frameSz`, is determined when the method is compiled and all its local declarations are processed; on the MIPS the run-time stack grows downward).

```
subi  $sp,$sp,frameSz# Push frame on stack
```

```

sw    $ra,0($sp)      # Save return address in frame
sw    $fp,4($sp)      # Save old frame pointer in frame
move  $fp,$sp        # Set $fp to access new frame
# Save callee-saved registers (if any) here
# Body of method is here
# Restore callee-saved registers (if any) here
lw    $ra,0($fp)      # Reload return address register
lw    $fp,4($fp)      # Reload old frame pointer
addi  $sp,$sp,frameSz# Pop frame from stack
jr    $ra             # Jump to return address

```

To translate an `invokevirtual` instruction, we must implement a dynamic dispatching mechanism. When a method `M` of class `C` is called using `invokevirtual`, we will be given, as the first parameter, an object of class `C` or any subclass of `C`. If `M` is redefined in the subclass (with exactly the same type as `M` in class `C`), the redefined version of `M` must be called. How do we know which version of `M` to execute?

The first parameter, compiled into a register, is a pointer to an object of class `C` or a subclass of `C`. To support garbage collection and heap management, each heap object has a type code as part of its header. This type code can be used to index into a dispatch table that contains the addresses of all methods the object contains. If we assign to each method a unique offset, we can use `M`'s offset, in the object's dispatch table, to choose the correct method to execute.

Fortunately, it is often the case that class `C` has no subclasses that redefine `M`. (e.g., if `C` or `M` is `private` or `final`). If dynamic loading of `C` is not possible, we can select `M`'s implementation at compile-time, and generate code to call it directly, without any table lookup overhead.

Example. As an example of the overall bytecode translation process, let us consider the following simple method, `stringSum`. This method sums integers from one to its parameter, `limit`, and returns a string representation of the sum:

```

public static String stringSum(int limit){
    int sum = 0;
    for (int i = 1; i <= limit; i++){
        sum += i;
    }
    return Integer.toString(sum);
}

```

The following bytecodes implement `stringSum`:

```

iconst_0      ; Push 0
istore_1      ; Store into variable #1 (sum)
iconst_1      ; Push 1
istore_2      ; Store into variable #2 (i)

```

```

        goto L2          ; Go to end of loop test
L1:    iload_1          ; Push var #1 (sum) onto stack
        iload_2          ; Push var #2 (i) onto stack
        iadd            ; Add sum + i
        istore_1         ; Store sum + i into var #1 (sum)
        iinc 2 1        ; Increment var #2 (i) by 1
L2:    iload_2          ; Push var #2 (i)
        iload_0          ; Push var #0 (limit)
        if_icmple L1     ; Goto L1 if i <= limit
        iload_1          ; Push var #1 (sum) onto stack
                                ; Call toString
        invokestatic    java/lang/Integer/toString(int)
        areturn         ; Return String reference to caller

```

In analyzing `stringSum`, we see references to three local variables (including the parameter). Adding in two words of control information, we conclude that a frame size of 5 words (20 bytes) is required. `limit` will be placed at offset 8, `sum` at offset 12, and `i` at offset 16.

In the code we generate below, we will follow the MIPS convention that one word function values, including object references, will be returned in register `$v0`. We will also exploit the fact that register `$0` always contains a zero value. The code we generate will begin with a method prologue (to push `stringSum`'s frame), then a line-by-line translation of its bytecodes, followed by an epilogue to pop `stringSum`'s frame and return to its caller.

```

        subi   $sp,$sp,20    # Push frame on stack
        sw    $ra,0($sp)    # Save return address
        sw    $fp,4($sp)    # Save old frame pointer
        move  $fp,$sp       # Set $fp to access new frame
        sw    $a0,8($fp)    # Store limit in frame
        sw    $0,12($fp)    # Store 0 ($0) into sum
        li    $t0,1        # Load 1 into $t0
        sw    $t0,16($fp)   # Store 1 into i
        j     L2            # Go to end of loop test
L1:    lw    $t1,12($fp)    # Load sum into $t1
        lw    $t2,16($fp)   # Load i into $t2
        add  $t3,$t1,$t2    # Add sum + i into $t3
        sw    $t3,12($fp)   # Store sum + i into sum
        lw    $t4,16($fp)   # Load i into $t2
        addi $t4,$t4,1      # Increment $t4 by 1
        sw    $t4,16($fp)   # Store $t4 into i
L2:    lw    $t5,16($fp)   # Load i into $t5

```

```

lw      $t6,8($fp)      # Load limit into $t6
sle     $t7,$t5,$t6     # set $t7 = i <= limit
bnez    $t7,L1          # Goto L1 if i <= limit
lw      $t8,12($fp)     # Load sum into $t8
move    $a0,$t8         # Copy $t8 to parm register
jal     String_toString_int_ # Call toString
                                # String ref now is in $v0
lw      $ra,0($fp)      # Reload return address
lw      $fp,4($fp)      # Reload old frame pointer
addi    $sp,$sp,20      # Pop frame from stack
jr      $ra             # Jump to return address

```

15.2 Translating Expression Trees

So far, we have concentrated on generating code from ASTs. We now focus on generating code from expression trees. In an expression tree interior nodes represent operators and leaves represent variables and constants. Many ASTs use this format for expressions, so much of this discussion applies to ASTs too.

An expression tree may be traversed and translated in many different orders. Normally, a depth-first, left-to-right traversal is used when translating expressions. A depth-first, left-to-right traversal always produces a *valid* translation. However alternative traversals may lead to better code (if exceptions, which must be tested in source order, are not a concern).

Consider the expression $(a-b) + ((c+d)+(e*f))$. The normal depth-first traversal first translates $(a-b)$, leaving its result in a register. Then $(c+d)+(e*f)$ is translated, requiring three registers (one to hold the first subexpression, and two more to evaluate the other subexpression). Thus a total of four registers is used. However, if the right subexpression, $(c+d)+(e*f)$, is evaluated first, only three registers are needed, because once this subexpression is computed its value can be held in one register, using the other two registers to compute $(a-b)$.

We now consider an algorithm that determines the minimum number of registers needed to evaluate any expression or subexpression. We ignore for the moment any special properties of operators, like associativity. The algorithm labels each node in a tree with the minimum number of registers needed to evaluate the subexpression rooted by that node. This labeling is called Sethi-Ullman numbering [Sethi Ullman 70]. Once the minimum number of registers needed for each expression and subexpression is known, we traverse the tree in a manner that generates *optimal* code (that is, code that minimizes register use and hence register spilling).

As we did in previous sections, we assume a MIPS-like machine model that requires that all operands be register-resident. The algorithm works in a bottom-up direction, first labeling leaves of the tree. All leaves are labeled with 1, since one

register is needed to hold a variable or constant. For interior nodes, which are assumed to be binary operators, the register requirements of the operands are considered. If both operands require r registers, the operator requires $r+1$ registers because an operand, once computed, will be held in a register. If the two operands require a different number of registers, the whole expression requires the same number of registers as the more complex of the two operands. (Evaluate the more complex operand first and save it in a register. The simpler operand needs fewer registers, and hence the registers used for the previous, more complex operand can be reused.) This analysis leads to the algorithm of Figure 15.1.

```

registerNeeds( T )
1.  if T.kind = Identifier or T.kind = IntegerLiteral
2.    then T.regCount  $\leftarrow$  1
3.    else registerNeeds(T.leftChild)
4.         registerNeeds(T.rightChild)
5.         if T.leftChild.regCount = T.rightChild.regCount
6.           then T.regCount  $\leftarrow$  T.rightChild.regCount+1
7.           else T.regCount  $\leftarrow$  max(T.leftChild.regCount,
                                         T.rightChild.regCount)

```

Figure 15.1 An Algorithm to Label Expression Trees with Register Needs

As an example of this algorithm, registerNeeds would label the expression tree for $(a-b) + ((c+d)+(e*f))$ as shown in Figure 15.2 (regCount for each node is shown at its bottom).

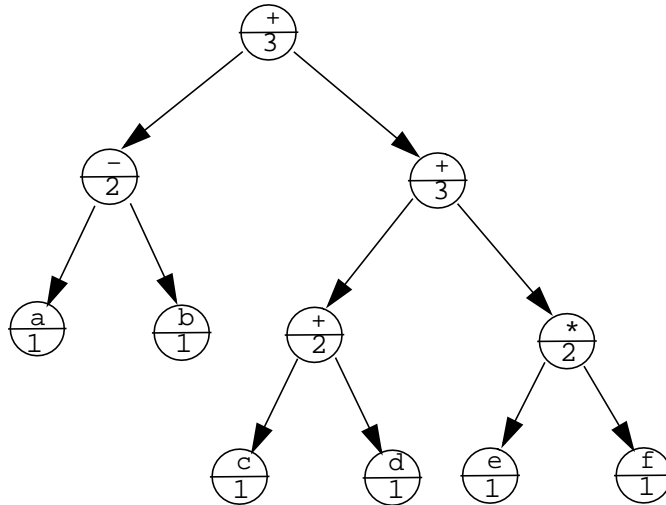


Figure 15.2 Expression Tree for $(a-b) + ((c+d)+(e*f))$ with Register Needs.

We can use the regCount labeling to drive a simple, but optimal, code generator, treeCG, defined in Figure 15.3. treeCG takes a labeled expression tree and a

list of registers it may use. It generates code to evaluate the tree, leaving the result of the expression in the first register on the list. If `treeCG` is given too few registers, it will spill registers, as necessary, into storage temporaries. (We use standard list manipulation functions, like `head` and `tail`, without defining them.)

```

treeCG( T, regList )
1.  r1 ← head(regList)
2.  r2 ← head(tail(regList))
3.  if T.kind = Identifier /* Load a variable */
4.    then generate(lw, r1, T.IdentifierName)
5.  elsif T.kind = IntegerLiteral /* Load a literal */
6.    then generate(li, r1, T.IntegerValue)
7.    else /* T.kind must be a binary operator */
8.      left ← T.leftChild
9.      right ← T.rightChild
10.     if left.regCount ≥ length(regList) and
11.        right.regCount ≥ length(regList)
12.       then /* Must spill a register into memory */
13.         treeCG(left, regList)
14.         temp ← getTemp() /* Get memory location */
15.         generate(sw, r1, temp)
16.         treeCG(right, regList)
17.         generate(lw, r2, temp)
18.         freeTemp(temp) /* Free memory location */
19.         generate(T.operation, r1, r2, r1)
20.       else /* There are enough registers; no spilling is needed */
21.         if left.regCount ≥ right.regCount
22.           then treeCG(left, regList)
23.                treeCG(right, tail(regList))
24.                generate(T.operation, r1, r1, r2)
25.           else treeCG(right, regList)
26.                treeCG(left, tail(regList))
                generate(T.operation, r1, r2, r1)

```

Figure 15.3 An Algorithm to Generate Optimal Code from Expression Trees

As an example, if we call `treeCG` with the labeled tree of Figure 15.2 and three registers, (`$10`, `$11` and `$12`), we obtain the following code sequence:

```

lw $10,c      # Load c into register 10
lw $11,d      # Load d into register 11
add $10,$10,$11 # Compute c + d into register 10
lw $11,e      # Load e into register 11
lw $12,f      # Load f into register 12
mul $11,$11,$12 # Compute e * f into register 11
add $10,$10,$11 # Compute (c + d) + (e * f) into reg 10

```

```

lw  $11,a      # Load a into register 11
lw  $12,b      # Load b into register 12
sub  $11,$11,$12 # Compute a - b into register 11
add  $10,$11,$10 # Compute (a-b)+((c+d)+(e*f)) into reg 10

```

treeCG illustrates nicely the principle of register targeting. Code is generated in such a way that the final result appears in the targeted register, without any unnecessary moves.

Because our simple machine model requires that all operands be loaded into registers, commutative operators (for which $exp1 \text{ op } exp2$ is identical to $exp2 \text{ op } exp1$) can't be exploited to improve code quality. However, most computer architectures aren't entirely symmetric. Thus in the MIPS R3000 architecture, some operations (like add and subtract) allow the right operand to be immediate. Immediate operands are small literal values included directly into an instruction; they need not be explicitly loaded into registers (see Exercise 8). For commutative operators, a small literal used as a left operand can be treated as if it were a right operand.

Some operations, like addition and multiplication are associative. Operands of an associative operator may be processed in any order. Thus, mathematically, $a+b+c$ and $c+b+a$ are identical. Regrouping operands of an associative operators can reduce the number of registers needed to evaluate an expression (see Exercise 9). For example, using registerNeeds we can establish that $(a+b)+(c+d)$ requires three registers whereas $a+b+c+d$ requires only two registers. Unfortunately, because of overflow and rounding issues, computer arithmetic is often *not* associative. Thus if a and b equal 1, c equals `maxint`, and d equals `-10`, $(a+b)+(c+d)$ will evaluate correctly, whereas $a+b+c+d$ may overflow. Many compilers reorder operands only when it is absolutely safe to do so.

15.3 Register Allocation and Temporary Management

An essential component of any code generator is its register allocator. Machine registers must be assigned to program variables and expressions. Since registers are limited in number, they must be reclaimed (reused) throughout a program.

A register allocator may be a simple “on the fly” algorithm that assigns and reclaims registers as code is generated. We'll consider on the fly techniques first. More thorough register allocators, that consider the register needs of an entire subprogram or program, will be considered next.

15.3.1 On the Fly Register Allocation

Most computers have distinct integer (general purpose) and floating register sets. In organizing our register allocator, we'll divide each register set into a number of classes:

- Allocatable registers
- Reserved registers

- Work registers

Allocatable registers are explicitly allocated and freed by compile-time calls to register management routines. While allocated, registers are protected from use by any but the “owner” of the register. Thus it is possible to guarantee that a register containing a data value will not be incorrectly changed by another use of the same register.

Requests for allocatable registers are usually *generic*; that is, requests are for any member of a register class, not for a particular register in that class. Usually any member of a register class will do. Further, generic requests eliminate the problem that arises if a particular requested register is already in use but many other registers in the same class are available.

A register, once allocated, must be freed when its assignment to a particular temporary is completed. A register is usually freed in response to an explicit directive issued by a semantic routine. This directive also allows us to mark the last use of a register as dead. This is valuable information because better code may be possible if the contents of a register need not be preserved.

Reserved and work registers, on the other hand, are never explicitly allocated or freed. Reserved registers are assigned a fixed function throughout a program. Examples include display registers, stacktop registers, argument and return value registers, as well as return address registers. Since the function of reserved registers is set by the hardware or operating system, and they are in use for all of a program or procedure, it is unwise to use such registers for other than their designated purpose.

Work registers may be used at any time by any code generation routine. Work registers may safely be used only in local code sequences, over which the code generator has complete control. That is, if we were generating code to do an indexing operation on an array (say, $a[i+j]$), it would be wrong to use a work register to hold the address of the array because computation of $i+j$ might change the work register. An allocatable register would, of course, be protected.

Work registers are useful in several circumstances:

- Sometimes we need a register for a very brief time. (For example, in compiling $a=b$ we load b into a register and then immediately store the register into a .) Using a work register saves the overhead of allocating and then immediately freeing a register.
- Many instructions require that their operands be in registers. What happens if *no* registers are free? Work registers are always free. If necessary, we can load values from memory into work registers, execute an instruction or two, then save needed values back into memory.
- We can pretend we have more registers than we really do. Such registers, sometimes called virtual registers or pseudo-registers, can be simulated by allocating them in memory and placing their values into work registers when they are used in instructions.

In general, reserved registers are identified in advance by hardware and operating systems conventions. Sometimes work registers are also established in

advance—if they aren't, choose three or four for this purpose. Remaining registers can be made allocatable. They will hold temporary values, and may also be used to hold frequently accessed variables and constants.

`getReg` and `freeReg`. To allocate and free registers, we'll create two subroutines, `getReg` and `freeReg`. `getReg` will allocate a single allocatable register and return its index. (If we have both integer and float registers, we will create `getReg` and `getFloatReg`.) A register allocated by a call to `getReg` is exclusively allocated to the caller until it is returned.

What happens if no more registers are available for allocation? In simple compilers we can simply terminate compilation with a message that the program requires more registers than are available. Modern computers routinely have at least 10–20 allocatable registers, so unless registers are used aggressively to hold program variables and constants, the chances of a “real life” program exhausting all allocatable registers is almost nil.

A more robust register allocator should not simply terminate when registers are exhausted. It can instead return pseudo-registers allocated in memory (in the frame of the procedure currently being translated). Pseudo-registers are encoded as integers greater than the indices of the real hardware registers. An array `regAddr[]` maps pseudo-registers into their memory addresses. Pseudo-registers are used exactly like real registers. The only difference is that when instructions using pseudo-registers are generated, they use work registers to move values back and forth from memory.

In some languages we may need to allocate temporaries in memory rather than registers. This may be because the temporary is too large to fit in a register (e.g., a struct returned by a function call) or because we need to be able to create a pointer to the temporary (most computers don't allow indirect references to registers). If we need storage based temporaries, we can create `getTemp` and `freeTemp` functions that essentially parallel `getReg` and `freeReg`. Temporaries allocated for a procedure are placed in the procedure's frame (essentially they are anonymous local declarations). Temporaries used by the main program may be allocated statically.

In some languages we may need to allocate temporaries whose size is not known at compile time. For example, if we use the `+` operator to concatenate C-style strings, then the size of `str1 + str2` will not in general be known until run-time. The temporary used to hold the result of such an expression can't be allocated statically or in a frame. Instead, we'd either push it onto the run-time stack or allocate space for it in the heap. Since pushing the stack is much more efficient (two or three instructions rather than hundreds), it is preferred. Note that when we return from the current procedure, the stack will be popped and the temporary *automatically* freed.

15.3.2 Register Allocation Using Graph Coloring

Using registers effectively is essential in generating efficient code for modern computers. We have already studied how to allocate registers in trees and “on the fly” as code is generated. In this section we address a greater challenge—how to allocate registers effectively throughout an entire procedure, function or main program. Since individual procedures and functions are the basic units of compilation in modern compilers, we have raised our sights from individual statements or expressions to entire procedure bodies.

Register allocation at the level of an entire subprogram is called global allocation, in contrast to allocation at the level of a single expression or basic block, which is termed local allocation. At the global level, a register allocator usually has many values that might profitably reside in registers (local and global variables, constants, temporaries containing available expressions, parameters and return values, etc.). Each value that might profitably reside in a register is called a register candidate; typically there are many more register candidates than there are registers.

Global register allocators do not usually allocate a register to a single value throughout the body of a subprogram. Rather, when possible, values that do not interfere with one another are assigned to the same register. Thus if variable *a* is used only at the top of a subprogram, and variable *b* is used only at the bottom on the subprogram, *a* and *b* may share the same register.

To enhance sharing, register candidates are divided into live ranges. A live range is the span of instructions in which a given value may be accessed, from its initial creation to its last use. For variables, a live range runs from its point of initialization or assignment to its last uses. For expressions and constants, a live range spans from their first to final use. Thus in Figure 15.4 variable *a* is broken into two separate and independent live ranges. Each is treated as a separate register candidate.

```
main() {
    a = f(x);           // Start of first live range
    print(a);          // End of first live range
    ....
    a = g(y)            // Start of second live range
    print(a);          // End of second live range
}
```

Figure 15.4 Example of Live Ranges

A live range can be readily computed using the SSA form described in Chapter 16, since each use of a variable is tied to *unique* assignment. Alternatively, one can avoid live range computation and simply treat each variable, parameter or constant as a distinct register candidate.

The Interference Graph. One of the central problems in global register allocation is deciding which live ranges may share the same register and which may not. A live range l is said to interfere with another live range m if l 's definition point (or beginning) is part of m 's range. This makes sense— l and m cannot share the same register if at the point l is first computed or loaded m is also in use.

To represent all the interferences in a subprogram (there normally are many), an interference graph is built. Nodes of the graph are the live ranges of the subprogram. An arc exists between live ranges l and m if l interferes with m or m interferes with l (the arc is undirected). Consider the simple procedure shown in Figure 15.5. It has four register candidates, a , b , c , and i . a , b , and i all interfere with one another; c interferes only with a . This interference information is concisely shown in the interference graph of Figure 15.6.

```

proc() {
    a = 100;
    b = 0;
    for (i=0;i<10;i++)
        b = b + i * i;
    print(a, b);
    c = 100;
    print(a*c);
}

```

Figure 15.5 A Simple Procedure with Candidates for Global Register Allocation.

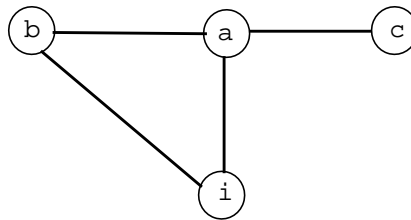


Figure 15.6 Interference Graph for procedure of Figure 15.5

With an interference graph, the problem of allocating registers is neatly reduced to a well-known problem—that of coloring the nodes of a graph. In the graph coloring problem the goal is to determine whether n colors suffice to color a graph given the rule that no two nodes linked by an arc may share the same color. This models exactly our problem of register allocation, where n is the number of registers we have available and each color represents a different register.

The problem of determining whether a graph is “ n -colorable” is NP-complete [Garey Johnson 79]. This means the best known algorithms that solve the problem have a time bound that is exponential in the size of the graph. As a result, register

allocators based on graph coloring normally use heuristics to solve the coloring problem.

We'll first consider an approach to register allocation using coloring devised by Chaitin [CAC 81, Cha 82]. Initially, the algorithm assumes that all register candidates can be allocated registers. This is often an impossible goal, so the interference graph is tested to see if it is n -colorable, where n is the number of registers available for allocation. If the interference graph is n -colorable, a register allocation is produced from the colors assigned to the interference graph.

If the graph is not n -colorable, it is simplified. A node (corresponding to a live range) is selected and spilled. That is, the live range is denied a register. Rather, whenever it is assigned to or used, it is loaded from or stored into memory using work registers, like the pseudo-registers of the previous section.

Since the live range that was spilled is no longer a register candidate it is removed from the interference graph. The graph is simpler and may now be n -colorable. If it is, our register allocation is successful—all remaining candidates can be allocated registers. If the graph still isn't n -colorable, we select and spill another candidate, further simplifying the graph. This process continues until an n -colorable graph is obtained.

Two questions arise. How do we decide if a graph is n -colorable? (Recall this is a very hard problem). If a graph isn't n -colorable, how do we choose the "right" register candidate to spill?

In testing for n -colorability, Chaitin made the following simple but powerful observation. If a node in the interference graph has fewer than n neighbors, that node can always be colored (just choose any color not assigned to any of its neighbors). Such nodes (termed unconstrained) are removed from the interference graph. This simplifies the graph, often making further nodes unconstrained. Sometimes all nodes are removed, demonstrating that the graph *is* n -colorable.

When only nodes with n or more neighbors remain, a node is spilled to allow the graph to be simplified. Chaitin suggests that in choosing a node to spill, two criteria be considered. First, the cost of spilling a node should be considered. That is, we compute the extra loads and stores that will have to be executed should a live range be spilled, weighted by the loop nesting level. Each level of loop nesting is arbitrarily assumed to add a factor of 10 to costs. Thus a live range in a single loop has its loads and stores multiplied by 10, a doubly nested loop multiplies loads and stores by 100, etc.

The second criterion Chaitin used is the number of neighbors a node has. The greater the number of neighbors a node has, the greater the number of interferences spilling the node removes. Chaitin suggests that the node with the smallest value of $cost/neighbors$ is the best node to spill. That is, the "ideal" node to spill is one that has a low spill cost and many neighbors, yielding a very small $cost/neighbors$ value.

Chaitin's algorithm is shown in Figure 15.7. As an example, consider the interference graph of Figure 15.6. Assume only two registers are available for allocation. Since c has only one neighbor, it is immediately removed from the graph and pushed on a stack for later register allocation. a , b and i all have two neigh-


```

GCRegAlloc( proc, regCount )
1.  ig ← buildInterferenceGraph(proc)
2.  stack ←  $\phi$ 
3.  while ig  $\neq$   $\phi$ 
4.      do if  $\exists d \in$  ig for which neighborCount(d) < regCount
5.          then ig ← ig - d
6.              push(d,stack)
7.          else Choose d such that
                   cost(d)/ neighborCount(d) is minimized
8.              ig ← ig - d
9.              Generate code to spill d's live range
10. while stack  $\neq$   $\phi$ 
11.     do d ← pop(stack)
12.     reg(d) ← any register not assigned to neighbors(d)

```

Figure 15.7 Chaitin's graph coloring register allocator.

bors. One will have to be spilled. *a* has a very low cost (3) because it is referenced only 3 times, all outside of the loop. *b* and *i* are used inside the loop and have much higher costs. Since all three nodes have the same number of neighbors, *a* is correctly chosen as the proper node to spill. After *a* is removed, *i* and *b* become unconstrained. When registers are assigned, *i* and *b* get different registers and *c* can be assigned either register. *a* gets no register. Rather, whenever it is used, it is loaded from or stored into a memory location, just like an ordinary variable.

Improvements to Graph Coloring Register Allocators. Briggs et al. [BCKT 89] suggest a number of useful improvements to Chaitin's approach. They point out that nodes with the smallest number of neighbors ought to be removed first from the interference graph. This is because nodes with few neighbors are the easiest to color and hence they ought to be processed last during the phase in which stacked nodes are popped and colored.

Another improvement follows from the observation that as nodes are removed to simplify the interference graph, they need not be spilled immediately. Rather, removed nodes should be stacked just like unconstrained nodes. When nodes are colored, constrained nodes *may* be colorable (because they happen to have neighbors that share the same color or happen to have neighbors that are also marked to be spilled). Constrained nodes that can't be colored are spilled only when we are sure they are uncolorable.

Register allocators need to handle two other problems. Assignments between register values are common. We would like to reduce register moves by assigning the source and target values in the assignment to the same register, making the assignment a trivial one. Moreover, architectural and operating system constraints sometimes force values to be assigned to specific registers. We'd like our allocator to try to choose register assignments that anticipate and adhere to predetermined register conventions.

To see how coloring allocators can handle register moves and preallocated registers, consider the following simple subprogram.

```
int doubleSum(int initVal, int limit({
    int sum = initVal;
    for (int i=1; i <= limit; i++)
        sum += i;
    return 2*sum; }
```

When this subprogram is translated, many short-lived temporary locations are created. Moreover, rules involving register allocation for parameters and return values are enforced. Prior to register allocation, `doubleSum` looks like the following

```
doubleSum(){
    initVal = $a0;    // First parm passed in $a0
    limit = $a1;     // Second parm passed in $a1
    sum = initVal;
    i = 1;
    temp1 = i <= limit;
    while (temp1) {
        temp2 = sum + i;
        sum = temp2;
        temp3 = i + 1;
        i = temp3;
        temp1 = i <= limit; }
    temp4 = 2 * sum;
    $v0 = temp4; // Return value register is $v0
}
```

The explicit use of register names in `doubleSum` represents nodes that must be allocated a particular register; these nodes are said to be precolored. If `a` and `b` are both allocated to registers, and we have the assignment `a = b`, an explicit register copy can be avoided if `a` and `b` are allocated to the same register. Values `a` and `b` will be automatically assigned to the same register if we coalesce their live ranges. That is, if we combine the nodes for `a` and `b` in the interference graph, then `a` and `b` *must* receive the same register.

When is coalescing `a` and `b` safe? At a minimum, they must not interfere. If they do interfere, then they are live at the same time and will need distinct registers. Even if `a` and `b` do not interfere, coalescing them may be problematic. The difficulty is that combining the live ranges of `a` and `b` will in general create a larger live range that is harder to color. We certainly don't want to spill a combined range when the individual ranges might have been individually colored.

To avoid problems in coloring coalesced interference graph nodes we can adopt a conservative approach. We will say a node in an interference graph has

significant degree if it has n or more neighbors (where n is the number of colors available). A node of significant degree may have to be spilled. A node that is insignificant (i.e., not significant) is always colorable. We can conservatively coalesce nodes a and b if the combined interference graph node has fewer than n significant neighbors. Why? Well, insignificant neighbors are always removed because they are trivially colorable. If the combined node has fewer than n significant neighbors, then, after insignificant neighbors are removed, the combined node will have fewer than n neighbors, so it too will be trivially colorable.

In our above `doubleSum` example, we have three values that must be register-resident (the two parameter values at the start, and the return value at the end). We have eight local variables and temporaries (`initVal`, `limit`, `i`, `sum`, `temp1`, `temp2`, `temp3` and `temp4`). We'll aim for a 4-coloring (a register allocation that uses 4 registers). Temporary `temp1` interferes with `i`, `limit` and `sum`, so we know that we can't use fewer than 4 registers without spilling.

We can coalesce `temp4` and `$v0`, guaranteeing that `2*sum` is computed into the return value register. We can coalesce `$a0` and `initVal`, allowing `initVal` to be accessed directly from `$a0` throughout the subprogram. Even more interestingly, we can then coalesce `initVal` and `sum`, allowing `sum` to use `$a0` too. Temporary `temp2` can also be coalesced with `sum`, allowing it too to use `$a0`. `limit` can be coalesced with `$a1`, allowing it to use `$a1` throughout the subprogram.

Temporary `temp3` can be coalesced with `i` since the combined node has fewer than 4 neighbors. Since neither `temp1` nor the combined `i` and `temp3` interfere with `$v0`, either of these can be assigned `$v0` to use. The other is assigned an unused register, for example `$t0`. The resulting register allocation, with register names replacing variables and temporaries, is shown below. Note that all register to register copies have been removed, and that only one register, beyond the pre-assigned ones, is used.

```
doubleSum() {
    $v0 = 1;
    $t0 = $v0 <= $a1;
    while ($t0) {
        $a0 = $a0 + $v0;
        $v0 = $v0 + 1;
        $t0 = $v0 <= $a1;
    }
    $v0 = 2 * $a0;
}
```

It is sometimes possible to coalesce interference graph nodes that have more than n significant neighbors. This is done by iterating between interference graph simplification and node coalescing [GA 96]. The resulting algorithm is very effective and is one of the simplest and most effective register allocators in current use.

15.3.3 Priority Based Register Allocation

Hennessey and Chow [90] and Larus and Hilfinger [86] suggest interesting alternatives to Chaitin’s graph coloring approach. After unconstrained nodes (which are trivially colorable) are removed from the interference graph, a priority is computed for each remaining node. This priority is similar to Chaitin’s cost estimate, except that it normalizes the cost using the size of the live range. That is, if two live ranges have the same cost, but one is smaller (in terms of the number of instructions it spans), the smaller live range ought to be given preference over the larger one. This makes sense—the smaller the live range is, the shorter is the span of instructions in which it “ties up” a register. The priority function recommended is $cost/size(live\ range)$. The greater the priority of a live range, the more likely it is to receive a register.

Another important difference is that when a node can’t be colored (because its neighbors have been allocated all the available colors), the node is split rather than being spilled. That is, if possible, the live range is divided into two smaller live ranges. Loads and stores are placed at the boundary of the split ranges, but each split range may be allocated a (possibly different) register. Because split ranges usually have fewer interferences than the original range, split ranges are often colorable when the original range is not.

There are many ways a live range may be split into smaller ranges. The following simple heuristic is often used:

1. Remove the first instruction of the live range (usually a load or computation), putting it into a new live range, NR .
2. Move successors to instructions in NR from the original live range to NR as long as NR remains colorable.

The idea is to break off at least one instruction, and then add instructions as long as the split range appears colorable. Instructions not split off remain in what’s left of the original live range, which may be split again. Single definitions or uses that can’t be colored are spilled.

A priority-based register allocator, `PriorityRegAlloc`, is shown in Figure 15.8. Reconsider the interference graph of Figure 15.6, assuming two registers, $\$r1$ and $\$r2$. Variable c is placed in unconstrained; it is trivial to color and will be handled after all other variables have been allocated registers. a , b and i are all placed in constrained. i has the highest priority for register allocation, since assigning it a register saves 51 loads and stores, and it spans only two statements. Assume it is assigned register $\$r1$. Variable b has the next highest priority (22 loads and stores saved). It is given $\$r2$. Variable a is the last candidate in constrained, but it can’t be colored. We split it into two smaller live ranges, $a1$ and $a2$. $a1$ is the single assignment at the top of the procedure. Range $a2$ spans the two print statements. $a1$ is effectively spilled since its range is a single instruction. $a2$ interferes with b but not i . Hence it receives $\$r1$. Finally, c receives $\$r2$.

```

PriorityRegAlloc( proc, regCount )
1.  ig ← buildInterferenceGraph(proc)
2.  unconstrained ← { n ∈ nodes(ig) | neighborCount(n) < regCount }
3.  constrained ← { n ∈ nodes(ig) | neighborCount(n) ≥ regCount }
4.  while constrained ≠ ∅
5.      do for c ∈ constrained such that not colorable(c) and canSplit(c)
6.          do c1, c2 ← split(c)
7.             constrained ← constrained - c
8.             if neighborCount(c1) < regCount
9.                 then unconstrained ← unconstrained + c1
10.            else constrained ← constrained + c1
11.            if neighborCount(c2) < regCount
12.                then unconstrained ← unconstrained + c2
13.               else constrained ← constrained + c2
14.               for d ∈ neighbors(c) such that d ∈ unconstrained
15.                  and neighborCount(d) ≥ regCount
16.                     do unconstrained ← unconstrained - d
17.                    constrained ← constrained + d
18.                /* At this point all nodes in constrained are colorable
19.                   or can't be split */
20.                Select p ∈ constrained such that priority(p) is maximized
21.                if colorable(p)
22.                    then color p
23.                    else spill p
24. color all nodes in unconstrained

```

Figure 15.8 A priority-based graph coloring register allocator.

15.3.4 Interprocedural Register Allocation

The global register allocators we have considered are limited by the fact that they consider only one subprogram at a time. Interprocedural interactions are ignored. Thus when a subprogram is called, either the caller or callee must save and restore any registers that both might use. When registers are used aggressively to hold a large number of variables, constants and expressions, saving and restoring common registers can make calls costly. Similarly, if a number of subprograms access the same global variable, each must load and later save the value when it is used.

Interprocedural register allocation improves overall register allocation by identifying and removing register conflicts across calls. Wall [86] considers interprocedural register allocation for architectures with a large number of registers. His goal is to assign registers so that caller and callee never use the same register. This guarantees that no saves or restores are needed during a call, making the call very inexpensive.

First, a priority estimate, similar to that of the previous section, is computed for each local variable or constant that might be kept in a register. These priorities are weighted by estimates of the execution frequency of each procedure. That is,

variables used by frequently executed subroutines have a much higher priority than those of infrequently executed subroutines. This is reasonable—we want to use registers most effectively in those subprograms that are executed most often. If procedure *a* calls *b*, the register allocator places the locals of *a* and *b* in different registers. Otherwise, a local of *a* and a local of *b* can share a common register. Groups of locals, one from each of a set of subprograms that can never be simultaneously active, are grouped together. The priority of a group is the sum of the priorities of all its member locals.

Registers are then “auctioned.” The group with the highest overall priority gets the first register. The next highest priority group gets the next register and so forth. Global variables are handled by placing them in singleton groups, with a priority equal to the total savings that result in all subprograms by having the global register-resident.

Wall found improvements of from 10% to 28% in execution speed, while from 83% to 99% of all dynamic memory references to data were removed. Since Wall’s scheme eliminates all saving and restoring, it works best when a large number of registers are available for allocation (52 in his tests). When fewer registers are available, saving and restoring must be included. Now the cost of giving a subprogram an extra register is compared with the benefit of having that register available for local use. If save-restore costs are less than the benefits, save and restore code is added.

When interprocedural effects are accounted for, it is possible to assign registers and position save-restore code in such a way that optimal register allocation is obtained [Kurlander Fischer 96]. The improvements in execution speed that result can sometimes be dramatic.

Some architectures, most notably the SPARC, provide register windows. When a call is made, the callee is provided its own “window” of registers, distinct from the caller’s register window. This reduces the cost of calls as saving and restoring of registers is done automatically. Register windows are allowed to partially overlap to facilitate parameter passing through registers.

15.4 Code Scheduling

We have already discussed the issues of instruction selection and register allocation in code generation. Modern computer architectures have introduced a new problem—that of ordering (or scheduling) the instructions that are generated. Most modern computer architectures are pipelined. This means that instructions are processed in stages, with an instruction progressing from stage to stage until it is completed. A number of instructions can be in different stages of execution at the same time. This is very important since instruction execution overlaps, allowing much faster execution speeds.

What happens if one instruction being executed needs a value produced by an earlier instruction that hasn’t yet completed execution? Normally this isn’t a problem—pipelines are designed to make results available as soon as possible. In a few cases however, a needed operand may not be available. Then the pipeline must be

stalled, delaying execution of an instruction (and its successors) until the needed value is available.

Most current pipelined architectures are delayed load. This means that a register value created by a load instruction is not available for use by the very next instruction. Rather it is *delayed* for one or more instructions. For example on a MIPS R3000 processor, loads are delayed by one instruction. This delay is needed to allow the processor's cache to be searched for the desired operand. Thus the following instruction sequence, though valid, would stall:

```
lw  $12,b          # Load b into register 12
add $10,$11,$12   # Add reg 11 and reg 12 into reg 10
```

Stalls are not inevitable after a load. If another instruction can be placed between a load and the instruction that uses the loaded value, instruction execution proceeds without delay. Thus the following instructions would be delay-free:

```
lw  $12,b          # Load b into register 12
li  $11,100        # Load 100 into register 11
add $10,$11,$12   # Add reg 11 and reg 12 into reg 10
```

The role of instruction scheduling is to order instructions so that stalls (and their delays) are minimized.

Code scheduling is normally done at the basic block level. A basic block is a linear sequence of instructions that contains no branches except at its very end. Instructions within a basic block are always executed sequentially, as a unit. During code scheduling all the instructions within a basic block are analyzed to determine an execution order that produces correct computations with a minimum of interlocks or delays. We'll consider a simple but effective postpass approach devised by Gibbons and Muchnick [1986].

Postpass code schedulers operate after code has been generated and registers have been chosen. They are very general because they can handle code generated by any compiler (or even hand-coded assembly language programs). However because instructions and registers have already been selected, they can't modify choices already made, even to avoid interlocks.

A code scheduler tries to move apart instructions that will interlock. However, instructions can't be reordered haphazardly—loads of a register must precede use of that register, and stores of a register must follow instructions that compute the register's value. We use a dependency dag to represent dependencies between instructions. Nodes of the dag are instructions that are to be scheduled. An arc exists between two instructions i and j if instruction i *must* be executed before instruction j . Thus arcs are added between instructions that load or compute a register and instructions that use or store that register. Similarly an arc is added between a load from memory location A and a subsequent store into location A . Also an arc is added between a store into location B and any subsequent load or store involving location B . In the case of aliasing, we make worst-case assumptions. Thus a load through a pointer P must precede a store into any location P might alias, and a store through P must precede any load or store involving a location P might alias.

As an example, assume we generate the MIPS code shown in Figure 15.9 for the expression $a = ((a * b) * (c + d)) + (d * (c + d))$.

```

1. lw    $10, a          6.  add   $10, $10, $12
2. lw    $11, b          7.  mul   $11, $11, $10
3. mul   $11, $10, $11   8.  mul   $12, $10, $12
4. lw    $10, c          9.  add   $12, $11, $12
5. lw    $12, d          10. sw    $12, a

```

Figure 15.9 MIPS code for $a = ((a * b) * (c + d)) + (d * (c + d))$

Figure 15.10 illustrates the corresponding dependency dag. Double-circled nodes are loads—the critical nodes in this example because they can stall.

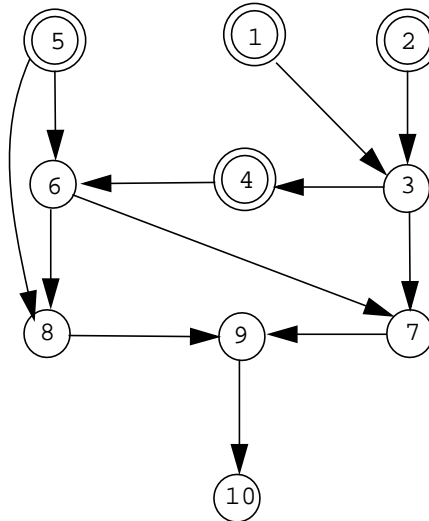


Figure 15.10 Dependency DAG for $a = ((a * b) * (c + d)) + (d * (c + d))$

Dependency dags have the property that any topological sort of the nodes represents a valid execution order. This is, as long as an instruction is scheduled before any of its successors in the dependency dag, it will execute properly. Any node that is a root of the dependency dag may be scheduled immediately. It is then removed from the dag, and again any root may be scheduled. Our goal in scheduling instructions will be to choose roots that avoid stalls. In fact the first rule in our scheduling algorithm is just that:

When choosing a root to schedule, choose one that *won't* be stalled by the most recently scheduled node.

Sometimes we can't find a root that doesn't stall its predecessor—not all instruction sequences are stall-free.

If we find more than one root that doesn't stall its predecessor, secondary criteria apply. We try to select the "nastiest" root—the one most likely to cause future stalls, or to complicate the scheduling process. Three criteria are considered, in decreasing order of importance

1. Does the root stall any of its successors in the dependency dag?
2. How many new roots will scheduling this root uncover?
3. What is the longest path from this root to a leaf of the dependency dag?

If a root can stall a successor, we want to schedule it immediately so that other roots can be scheduled before the successor, avoiding a stall. If we schedule a root that exposes other new roots, we increase the range of choices available to the scheduler, simplifying its task. If we schedule a root with a long path to a leaf, we are attacking a "critical path," a long instruction sequence that allows the scheduler few choices in reordering instructions.

In our scheduling algorithm, `scheduleDag`, we'll use an operation `select` that takes a set of root nodes of the dependency dag, and a criterion. `select` will choose the nodes in the root set that meet the criterion, as long as the set selected is non-empty. That is, if no node in the set meets the criterion, `select` will return the entire input set. The reason for this is that a criterion that rejects all nodes is useless since our goal is to choose *some* node to schedule.

For example, `select(nodeSet, "Has the longest path to a leaf")` selects those nodes in `nodeSet` with the greatest distance to a leaf (several nodes may be selected if they all share the same maximum distance to a leaf). However, `select(nodeSet, "Can stall some successor")` would return all of `nodeSet` if no member of the set had a successor that it stalled. Once we have refined a `nodeSet` to a single node, further applications of `select` are unnecessary; they will have no effect.

```

scheduleDag( dependencyDag )
1.  while candidates ← roots(dependencyDag) ≠ ∅
2.      do select(candidates, "Is not stalled by last instruction generated")
3.         select(candidates, "Can stall some successor")
4.         select(candidates, "Exposes the most new roots if generated")
5.         select(candidates, "Has the longest path to a leaf")
6.         Let Inst ∈ candidates
7.         Schedule Inst as next instruction to be executed
8.         dependencyDag ← dependencyDag - Inst

```

Figure 15.11 An Algorithm to Schedule Code from a Dependency Dag.

The complete definition of `scheduleDag` is shown in Figure 15.11. An example, consider the dependency dag of Figure 15.10. The code originally generated (Figure 15.9) contains two stalls (after instructions 2 and 5). The initial set of roots is 1, 2 and 5, all load instructions. All roots can stall a successor instruction and none expose a new root if scheduled. Both 1 and 2 have the longest path to a leaf, so 1 is arbitrarily chosen and scheduled. The root set is now 2 and 5. Instruction 2 is chosen because it exposes a new root, 3. Next 5 is chosen because it can

stall a successor. Instructions 3, 4 and 6 are chosen next, as they form, in turn, singleton root sets. Instructions 7 and 8 are the new root set; 7 is arbitrarily chosen, then 8, 9 and 10. The code we produce is shown in Figure 15.12.

```

1. lw    $10,a          6.  add   $10,$10,$12
2. lw    $11,b          7.  mul   $11,$11,$10
3. lw    $12,d          8.  mul   $12,$10,$12
4. mul   $11,$10,$11    9.  add   $12,$11,$12
5. lw    $10,c          10. sw    $12,a

```

Figure 15.12 Scheduled MIPS code for $a = ((a*b) * (c+d)) + (d*(c+d))$

15.4.1 Improving Code Scheduling

The code shown in Figure 15.12 is not perfect. A stall still occurs after the fifth instruction. In fact, using just three registers a stall *can't* be avoided. It is shown in [Kurlander et al. 95] that sometimes an additional register is needed to avoid all stalls. One way to improve the code produced by `scheduleDag` is to reallocate registers in the initial code sequence, using an extra register beyond the original allocation.

To do this, we find instructions that stall and try to move them “up” in the instruction sequence. If we can't move a stalling instruction earlier because it assigns to a register used by the preceding instruction, we reallocate the register assigned to by the stalling instruction to be a register unused by the preceding instruction. Because we've added an extra register, we can always find an unused register, and move the stalling instruction at least one position earlier in the execution sequence.

For example, reconsidering Figure 15.12, instruction 5 (a load) stalls because \$10 is used in instruction 6. We can't move instruction 5 up because instruction 4 uses a previous value of \$10, loaded in instruction 1. If we add an additional register, \$13, to our allocation, we can load it in instruction 5 (taking care to reference \$13 rather than \$10 in instruction 6.) Now instruction 5 can be moved earlier in the sequence, avoiding a stall. The resulting delay-free code is shown in Figure 15.13.

```

1. lw    $10,a          6.  add   $10,$13,$12
2. lw    $11,b          7.  mul   $11,$11,$10
3. lw    $12,d          8.  mul   $12,$10,$12
4. lw    $13,c          9.  add   $12,$11,$12
5. mul   $11,$10,$11    10. sw    $12,a

```

Figure 15.13 Delay-free MIPS code for $a = ((a*b) * (c+d)) + (d*(c+d))$

It is evident that there is a tension between code scheduling, which tries to increase the number of registers used (to avoid stalls), and code generation, which

seeks to reduce the number of registers used (to avoid spills and make registers available for other purposes). An alternative to postpass code scheduling is an *integrated approach* that intermixes register allocation and code scheduling.

The Goodman Hsu [1988] algorithm is a well-known and widely used integrated register allocator and code scheduler. As long as registers are available, it uses them to improve code scheduling by loading needed values into distinct registers. This allows loads to “float” to the beginning of the code sequence, eliminating stalls in later instructions that use the loaded values. When registers grow scarce, the algorithm switches emphasis, and begins to schedule code to free registers. When sufficient registers are available, it resumes scheduling to avoid stalls. Experience has shown that this approach balances nicely the need to use registers sparingly and yet avoid stalls whenever possible.

15.4.2 Global and Dynamic Code Scheduling

Although we have focused on code scheduling at the basic block level, it is possible to schedule code at the global level [Bernstein & Rodeh 1991]. Instructions may be moved upward, past the beginning of a basic block to predecessor blocks in the control flow graph. We may need to move instructions out of a basic block because basic blocks are often very small—sometimes only an instruction or two in size. Moreover, certain instructions, like loads and floating point multiplies and divides, can incur long latencies. For example, a load that misses in the primary cache may stall for 10 or more cycles; a miss in the secondary cache—to main memory—can cost 100 or more cycles.

As a result, code schedulers often seek to move loads as early as possible in an instruction sequence. There are several complicating factors, however. To what predecessor block should we move an instruction? Ideally, to a predecessor that is control equivalent; that is, a predecessor that will be executed if, and only if, the current block is. An example of this is moving an instruction that follows an `if` statement to a position that precedes the `if` (and thereby past both arms of the `if`). An alternative is to move an instruction to a block that dominates it (that is, to a block that is a necessary predecessor). Now however, the moved instruction may be speculative—it may be executed unnecessarily on some execution paths. Thus if an instruction is moved from a `then` part to a position above the `if`, the instruction will be executed even when the `else` part is selected. Speculative instructions may waste computational resources, by executing useless instructions. What’s worse, if a speculative instruction faults (e.g., a load through a null or illegal pointer), a false run-time error may be created.

Even if we can move an instruction freely upward, how far should we move it? If we move the instruction too far forward, it will “tie up” a register for an extended period, making register allocation harder and less effective. Some architectures, like the DEC alpha, provide a prefetch instruction. This instruction allows data to be loaded into the primary cache in advance, reducing the chance that a register load will miss. Again, placement of preloads is a tricky scheduling issue. We want to preload early enough to hide the delays incurred in loading the

cache. But, if we preload too early, we may displace other useful cache data, causing cache misses when *these* data are used.

A number of recent computer architectures (MIPS R10000, Intel Pentium Pro) have included a sophisticated dynamic scheduling facility. These designs, sometimes called out of order architectures, delay instructions that aren't ready to execute, and dynamically choose successor instructions that are ready to execute. These designs are far less sensitive to compiler-generated code schedules. In fact, dynamically scheduled architectures are particularly effective in executing old programs ("dusty decks") that were created before code scheduling was even invented.

Even with dynamically scheduled architectures compiler-generated code scheduling is still an important issue. Loads, especially loads that frequently miss in the primary cache, must be moved early enough to hide the long delays a cache miss implies. Even the best current architectures can't look dozens or hundreds of instructions ahead for a load that might miss in the cache. Rather, compilers must identify those instructions that might incur the greatest delays and move them earlier in the instruction sequence.

15.5 Automatic Instruction Selection

An important aspect of code generation is instruction selection. After a translation for a particular construct is determined, the machine level instructions that implement the translation must be chosen. Thus if we decide to implement a `switch` statement using a jump table (see Section 13.1.5), instructions that index into the jump table and then do an indirect jump must be generated.

Often several different instruction sequences can implement a particular translation. Even something as simple as $a + 1$ can be implemented by loading 1 into a register and generating an add instruction, or by generating an increment or add immediate instruction. We normally want the smallest or fastest instruction sequence. Thus an add immediate, because it avoids an explicit load, is preferred.

In simple RISC architectures, the choice of potential instruction sequences is limited because almost all operands must be loaded into registers before they can be used (immediate operands being a notable exception). Further, the variety of addressing modes provided is also spartan—often only absolute and indexed addresses are allowed.

Older architectures, like the Motorola 680x0 and Intel x86, are much more elaborate. Many different operation codes are provided, and a wide variety of addressing modes are available. Operands need not always be loaded into registers, addressing modes can fetch operands indirectly and can increment and decrement registers. Different register classes (e.g., address registers and data registers) are used in different instructions (in a non-interchangeable manner) and particular registers are sometimes "wired into" certain instructions.

For very complex architectures, a way of systematizing and automating instruction selection is vital. Even for simpler architectures, it may be necessary to "extend" a code generator when a successor architecture is introduced. Very

ambitious compilers may aim to compile into more than one target architecture, mandating alternative instruction sequences for different target machines.

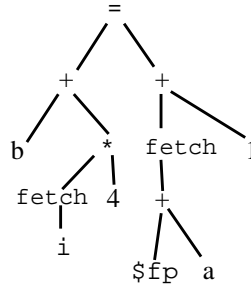


Figure 15.14 A Low-level IR Representation of $b[i]=a+1$

Instruction selection is often simplified by translating source language constructs into a very low-level tree-structured intermediate representation (IR). In this IR, leaves represent registers, memory locations, or literal values, and internal nodes represent basic operations on operand values. Detailed data access patterns and manipulations are exposed. (For an excellent example of a low-level tree-structured IR see [Appel 87]) Consider the statement $b[i]=a+1$, where b is an array of integers, i is a global integer variable, and a is a local variable accessed through the frame register, $\$fp$. The statement's tree-structured IR is shown in Figure 15.14. Note that leaves corresponding to identifiers are their addresses (if globals) or offsets (if locals). Explicit memory fetches (using the `fetch` operator) are shown, as is the multiply by 4 needed to build a valid word address for an element of an array of integers.

A tree-structured IR may also be used to define the effect of each instruction of a computer. A tree defines the computation performed by the instruction as well as the kind of value it produces. This is illustrated in Figure 15.15 in which tree-structured patterns (or productions) are used to define valid IR trees.

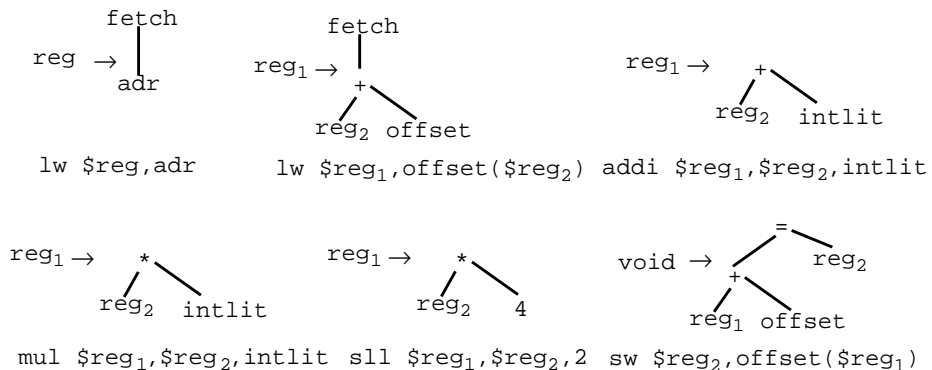


Figure 15.15 IR Tree Patterns for Various MIPS Instructions.

Now instruction selection for a given IR tree becomes a matter of matching instruction patterns against the generated IR such that the IR tree is covered (parsed) with adjacent patterns. That is, we find a subtree in the IR translation that matches exactly the pattern for some instruction. That subtree is then replaced with the pattern's left hand side. The process is repeated until the entire IR tree is reduced to a single node. This is very similar to ordinary bottom-up parsing (Chapter 6).

As instruction patterns are matched, their corresponding machine language instructions are generated. Registers can be allocated “on the fly” using the techniques of Section 15.3.1. Alternatively, pseudo-registers can be allocated as code is generated, and then later mapped to real registers using the graph coloring techniques of Section 15.3.2.

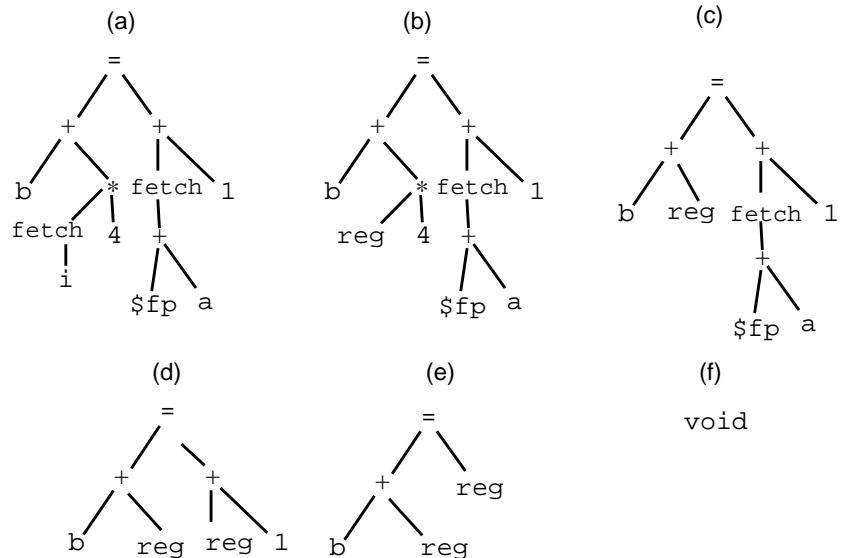


Figure 15.16 Instruction Selection Using Patterns.

As an example, reconsider the IR tree corresponding to $b[i]=a+1$ (Figure 15.16 (a)). We first match a load of i (b). Next, a multiply by 4 is matched (c). Then an indexed load is generated for a (a local variable), (d). Finally, an add immediate (e) and a store instruction, (f), reduce the IR tree to `void`. The instructions generated (assuming calls to `getReg` and `freeReg` as code is generated) are shown in Figure 15.17.

15.5.1 Instruction Selection Using BURS

It is often the case that more than one instruction sequence can implement the same construct. In terms of IR trees, different reductions of the same tree, yielding

```

lw      $t1,i
mul     $t1,$t1,4
lw      $t2,a($fp)
addi   $t2,$t2,1
sw      $t2,b($t1)

```

Figure 15.17 MIPS code for $b[i]=a+1$

different instruction sequences, may be possible. How can we choose the instruction sequence to be generated?

A very elegant approach involves assigning costs to instruction patterns. The cost of an instruction is set when a code generator is built. This cost may be size of an instruction, or its execution speed, or the number of memory references the instruction makes, or any criterion that measures how “good” an instruction is. When given a choice, we’ll prefer a cheaper instruction over a more expensive one.

Now matching of instruction patterns to an IR tree is generalized so that a least-cost cover is obtained. That is, the pattern matcher guarantees that the matches it selects have the lowest possible cost. Thus using the measure of quality selected when the code generator was built, the best possible instruction sequence is generated.

To guarantee that a least-cost cover of an IR tree is found, we use dynamic programming. Starting at the leaves of the tree, we mark each leaf with the lowest cost possible to reduce the leaf to each of the nonterminals. (Nonterminals, as in context-free productions, are the symbols that appear on the left-hand side of instruction patterns). Next interior nodes just above the leaves are considered. Each instruction pattern that correctly matches the interior node and has the correct number of children is considered. The cost of the pattern plus the costs of the node’s children are considered. The node is marked with the cheapest cost possible to reduce it to each nonterminal. We continue traversing the IR tree, until the root node is reached. The lowest cost found to reduce it to any nonterminal is selected as the best (least-cost) cover.

IR trees for a large program or subroutine can easily comprise tens of thousands of nodes. The extensive processing needed for each node would appear to make least-cost instruction selection using patterns a very slow process. Happily this is not the case.

An approach based on BURS theory (Bottom Up Rewrite Systems) [PG 88] allows very fast instruction selectors (and code generators) to be built. Code generators built using BURS theory can be extremely fast because all dynamic programming is done in advance when a special BURS automaton is built. During compilation it is only necessary to make two traversals of the IR tree: one bottom-up traversal to label each node with a state that encodes all optimal matches and a second top-down traversal that uses these states to select and generate code. It has been reported that careful encodings can produce an automaton that executes fewer than 90 RISC instructions per node to do both traversals.

The automaton that labels the tree is a simple finite state machine, similar to that used in shift-reduce parsers (Chapter 6). A bottom-up walk of the tree is performed, and the label for any given node is determined by a table lookup given the operator at the node and the states that label each of its children. The automaton that emits code is equally simple in design. The code to be emitted is determined by the state that labels a node and by the nonterminal to which that node should be reduced—another table lookup.

As an example, the instruction patterns of Figure 15.15 would all be given a cost of 1 except for `mul`, which would be given a cost of 3. This is because `mul` is actually implemented by the MIPS assembler using three hardware instructions, whereas all the other instructions are implemented using a single instruction. Returning to the example of Figure 15.14, all the leaves would be labeled with a state indicating that no reductions of leaf nodes are possible—the leaves must all be matched directly. Visiting `i`'s parent with its state, the `fetch` would be labeled with a state indicating that application of an `lw` pattern is possible, at a cost of 1. Going to its parent (`a * operator`), the state reached would show that although two reductions are possible (patterns for both `mul` and `sll` match); `sll`, being cheaper, will apply. That is, the instruction selector has recognized a well-known trick—multiplication by a power of two can often be implemented more efficiently by doing a left shift rather than an explicit multiply.

Continuing, the rest of the nodes are labeled, with the remaining matches being identical to those illustrated in Figure 15.16. The state labeling the root tells us that the final instruction to be generated (to implement the assignment) will be a `sw`. The two subtrees are visited to generate the instructions needed to implement them. We therefore generate the root's instruction after returning from recursive visits to both children, guaranteeing that the store's operands are computed prior to its execution. We generate the code shown in Figure 15.18.

```
lw      $t1,i
sll     $t1,$t1,2
lw      $t2,a($fp)
addi    $t2,$t2,1
sw      $t2,b($t1)
```

Figure 15.18 Improved MIPS code for `b[i]=a+1`

Two difficulties arise in creating a BURS-style code generator: efficiently generating the states and state transition tables (because *all* potential dynamic programming decisions are done at table generation time, they must be done efficiently) and creating an efficient encoding of the automata for use in a compiler. Fraser and Henry discuss a solution to the encoding problem in [FH 91]. Proebsting created BURG, [Pro95], a simple and efficient tool for generating BURS-style code generators. Using a very clean implementation and ingenious state elimination techniques, least-cost code generators for a variety of architectures can be created in a few seconds.

15.5.2 Instruction Selection Using Twig

Other code generation systems based on tree pattern matching and dynamic programming have been developed. They differ primarily from BURS in how they do tree pattern matching and in the fact that they do dynamic programming at compile-time rather than compile-compile time.

Aho, Ganapathi, and Tjiang [AGT 89] created a tree manipulation language and system called Twig. Given a specification of tree patterns and associated costs, Twig generates a top-down tree automaton that will find the least-cost cover of a subject tree. Twig uses fast top-down Hoffmann-O'Donnell [HO 82] pattern matching in parallel with dynamic programming to find the least-cost cover.

Starting at the root of possible instruction trees, paths to each of the tree's children are traced. Whenever a such a path is correctly traced, a counter is incremented. When the counter equals the number of children a pattern tree has, a potential match is recognized. Using costs and dynamic programming, the least-cost cover for an entire IR tree can be found.

The costs associated with patterns in Twig are more general than those afforded by any BURS system. Twig may compute the cost of a pattern dynamically—depending on semantic information available at compile-time. This flexibility further allows Twig to abort certain matches if semantic predicates are not satisfied. Thus, the applicability of Twig's patterns is context sensitive. BURS does not have this flexibility since all costs must be fixed prior to compilation to allow precomputation of dynamic programming decisions. The great advantage of BURS is its speed. *All* possible matches are anticipated in advance and tabulated. Twig must recognize partial matches and update counters as instruction selection proceeds. Given the huge IR trees that often need to be translated, even a little extra processing at each node can represent a significant slowdown.

15.5.3 Other Approaches

One of the first instruction selection techniques based on tree rewriting was that of Cattell [Cat 80]. First the effect of each instruction was described, using a register-transfer notation. Then a code generator “discovered” appropriate code sequences by matching instructions against IR trees. That is, the code generator explored ways to decompose an IR tree into combinations of special primitive trees, using backtracking if necessary. Because this process could be very slow, a catalog of the tree patterns that are implemented was precomputed. At compile-time this catalog was searched to find available instruction sequences.

Glanville and Graham [GG 78] observed that the problem of matching code templates against an IR tree is very similar to the problem of matching productions against a token sequence during parsing. They cleverly reformulated the template-matching problem in context-free parsing terms. Using standard shift-reduce parsers, augmented to handle multiple template matches, instruction selection could be automated.

A limitation of the Graham-Glanville approach is that it is purely syntactic. It simply matches, in a context-free manner, sequences of symbols. Ganapathi and

Fischer [GF 85] suggested adding attributes to code templates. Attributes allow types, sizes and values to influence instruction selection.

The Back End Generator (BEG) [ESL 89] finds a least-cost cover of the tree using dynamic programming techniques that are essentially identical to Twig's. Like Twig, BEG can guard patterns with semantic predicates. A BEG specification, in addition to having instruction patterns, includes a description of the register set of the target machine. This specification automatically generates the register allocator. Experiments show code quality and code generation times to be comparable to handwritten code generators.

Fraser, Hanson, and Proebsting [FHP 92] developed a code-generator generator based on naive pattern matching and dynamic programming. This system, *iburg*, maintains the same interface as BURG. Although *iburg* code generators are slower than those generated by BURG, *iburg* presents a simple and efficient framework for the development of pattern-based code generators.

15.6 Peephole Optimization

To produce high-quality code, it is necessary to recognize a multitude of special cases. For example, it is clear we would like to avoid generating code for an addition of zero to an operand. But where should we check for this special case? In each semantic routine that might generate an add? In each code-generation routine that might emit an add instruction?

Rather than distribute knowledge of special cases throughout semantic or code-generation routines, it is often preferable to utilize a distinct peephole optimization phase that looks for special cases and replaces them with improved code. Peephole optimization may be performed on ASTs, IR trees [Tan 82] or generated code [McK 65]. As the term “peephole” suggests, a small window of two or three instructions or nodes is examined. If the instructions in the peephole match a particular pattern, they are replaced with a replacement sequence. After replacement, the new instructions are reconsidered for further optimization.

In general, we represent the collection of special cases that define a peephole optimizer as a list of pattern-replacement pairs. Thus, *pattern* \Rightarrow *replacement* means that if a instruction sequence or tree matching the pattern is seen, it is replaced with the replacement sequence. If no pattern applies, the code sequence is unchanged. Clearly the number of special cases that might be included is unlimited. We will illustrate where peephole optimization can be employed, and the kinds of optimizations that can be realized.

15.6.1 Levels of Peephole Optimization

In general there are three places where peephole optimization may profitably be employed. After parsing and typechecking, a program is represented in AST form. Here peephole optimization may be used to optimize the AST, recognizing special cases at the source level that are independent of how a construct is translated, or the code that is generated for it.

After translation, a program is represented in an IR or bytecode form. Here peephole optimization can recognize optimizations that simplify or restructure an IR tree or bytecode sequence. These optimizations are independent of the actual target machine or the exact code sequences used to implement an IR tree or bytecodes.

Finally, after code generation peephole optimization can replace pairs or triples of target machine instructions with shorter or simpler instruction sequences. At this level, the optimization is highly dependent on the details of a machine's instruction set.

AST Level Optimizations. In Figure 15.19 we illustrate optimizations that can simplify or improve an AST representation of a program. In (a) an IF whose condition is always true is replaced with the body of the conditional. In (b) and (c), expressions involving constant operands are “folded” (replaced with the value of the expression). This folding optimization can expose other optimizations (such as the conditional replacement optimization of (a)).

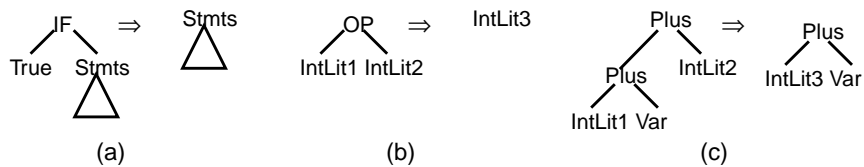


Figure 15.19 AST Level Peephole Optimization.

Optimizations at the AST level can conveniently be implemented using a tree rewriting tool like BURS. Source patterns are first recognized and labeled. Then during the “processing” traversal, trees can be rewritten into the target form. If necessary, an AST can be traversed several times, so that rewritten ASTs can be matched and transformed several times.

IR Level Optimizations. As illustrated in Figure 15.20, a variety of useful optimizations can be performed at the IR level. In (a) and (b), constant folding is specified. Since some arithmetic operations are exposed only after translation (e.g., indexing arithmetic), folding can be done at both the AST and IR levels. In (c), multiplication by a power of 2 is replaced with a left shift operation. In (d) and (e) identity operations are removed. In (f) the commutativity of addition is exposed, and in (g) addition of a negative value is transformed into subtraction.

Transformations on IR trees can be conveniently implemented using a tool like BURS.

As illustrated in Figure 15.21, optimizations corresponding to those of Figure 15.20 can be applied to a bytecode representation of a program. This level of optimization may be appropriate if bytecodes are later expanded into target machine code. Alternatively, the machine-level optimizations described in the next section

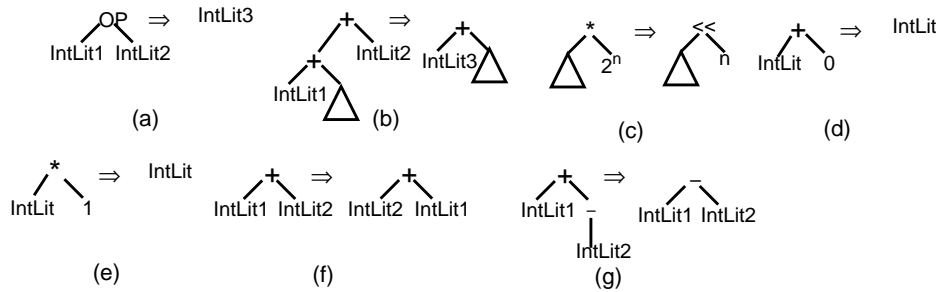


Figure 15.20 IR Level Peephole Optimizations.

may be applied to bytecodes, since bytecodes share much of the structure of conventional machine code.

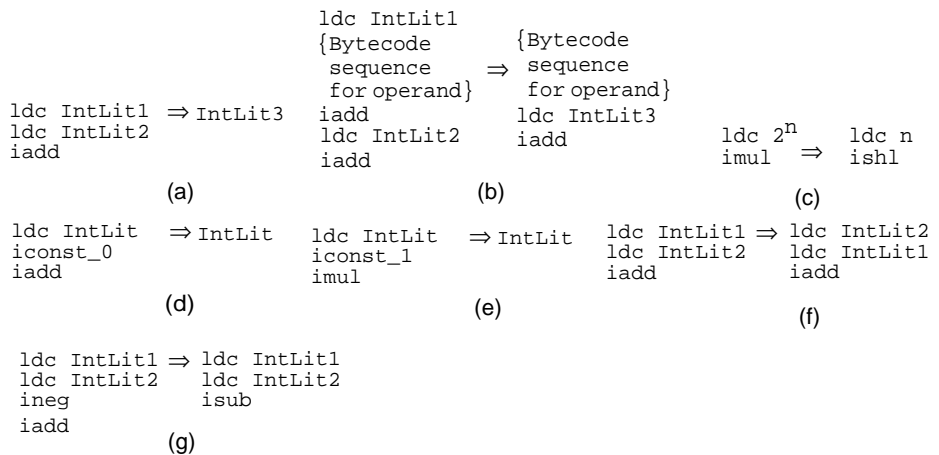


Figure 15.21 Bytecode Level Peephole Optimizations.

Code Level Optimizations. Figure 15.22 illustrates some simple peephole optimizations performed after code generation. In (a) a conditional branch around an unconditional branch is replaced with a single conditional branch (with the sense of the test inverted). In (b), a branch to the next instruction is removed (this is sometimes generated when a then or else part of an if is null). A branch to a second branch can be collapsed to a direct branch to the final target (c). In (d) a move from a register to itself is suppressed (this sometimes happens when a special register, like a parameter register, is loaded with a value that already is in the correct register). In (e) a register is stored into a location and then that same register is immediately reloaded from the same location; the load is unnecessary and may be deleted.

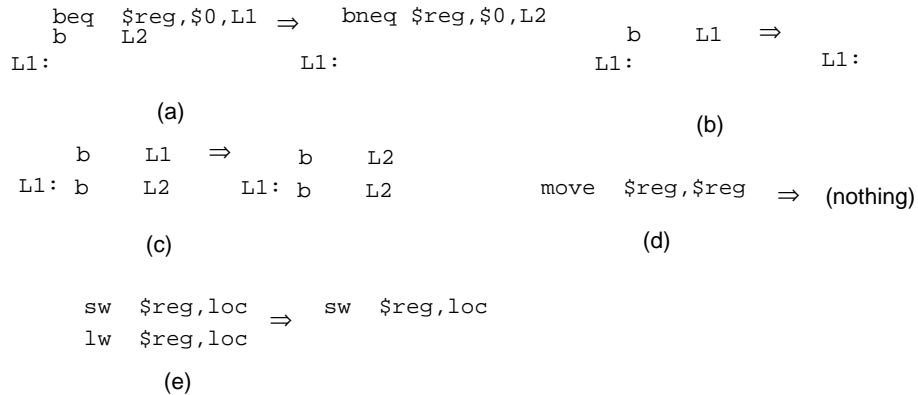


Figure 15.22 Code Level Peephole Optimizations.

More elaborate architectures present additional opportunities for peephole optimization. If a special increment or decrement instruction is available, it can replace an ordinary add immediate (which usually is longer and a bit slower). If auto-increment or auto-decrement addressing modes are available, these can be used to “hide” an explicit increment or decrement of an index. Some architectures have a special “loop control” instruction that decrements a register and conditionally branches if it is zero.

Recognizing replacement patterns must be done quickly, if peephole optimization is to be fast. Operator-operand combinations are hashed to applicable patterns. Also, the size of a peephole window is normally limited to two or three instructions. Using a careful hashed implementation, speeds of several thousand instructions per second have been achieved [DF 84].

The concept of analyzing physically adjacent instructions has been generalized to logically adjacent instructions [DF 82]. Two instructions are logically adjacent if they are linked by flow of control or if they are unaffected by intervening instructions. (The “branch chain” of Figure 15.22 (c) is a good example of this.) By analyzing logically adjacent instructions it is possible to remove jump chains (jumps to jump instructions) and redundant computations (for example, unnecessarily setting a condition code). Detecting logical adjacency can be costly, so care is required to keep peephole optimization fast.

15.6.2 Automatic Generation of Peephole Optimizers

In [FD 80] ways of automating the creation of peephole optimizers are discussed. The idea is first to define the effect of target machine instructions at the register-transfer level. At this level, instructions are seen to modify primitive hardware locations, including memory (represented as a vector M), registers (represented as a vector R), the PC (program counter), various condition codes, and so

on. A target machine instruction may have more than one effect, and its definition at the register-transfer level may include more than one assignment.

The peephole optimizer (PO) operates by considering pairs of instructions, expanding them to their register-transfer level definitions, simplifying the combined definitions, and then searching for a *single* instruction that has the same effect as the combined pair.

To be applicable, an instruction must perform all the register transfers of the combined instructions. It may also do other register transfers as long as these are dead (and therefore have no effect on subsequent computations). Thus an instruction may set a condition code, even if this is not wanted, as long as the updated condition code is not referenced by later instructions.

Instruction pairs that start with a conditional branch get special treatment. In particular, the second instruction is prefixed with a conditional representing the negation of the original condition (the only way the second instruction is executed is if the conditional branch fails). An unconditional branch is paired with its target instruction. This pairing often allows jump chains (a jump to another jump) to be collapsed. Note, however, that instruction pairs with the second instruction labeled are not optimized. This situation is needed to make jumps to such labels work correctly. However, if all references to a label are removed by the PO, then the label itself is also removed, possibly allowing new optimizations to be discovered.

The analysis and simplification of the instructions just described are not actually done during compilation because this would be far too slow. Rather, representative samples of actual programs are analyzed in advance, and the most common peephole optimizations are stored in a table. During compilation, this table is consulted to determine if the instructions currently in the peephole may be optimized.

Exercises

1. Consider the following Java function:

```
public static int fact(int n){
    if (n == 0)
        return 1;
    else return n*fact(n-1); }
```

Show the JVM bytecodes that would be generated for this method. Explain how these bytecodes would be translated to target machine code using the techniques of Section 15.1.

2. Recall that in Section 13.1.5 switch statements were translated using either a `tableswitch` or `lookupswitch` bytecode. Using either the MIPS architec-

ture or your favorite processor architecture, explain how the `tableswitch` and `lookupswitch` can be efficiently translated into machine-level instructions.

3. On many processors certain registers *must* be used to hold a parameter to a subprogram or a return value from a function. Suggest how the techniques of Section 15.1 could be extended so that when bytecodes are translated, parameters and return values are computed directly into the required register (without any unnecessary register to register moves).
4. Recall that a key to generating efficient target-machine code from bytecodes is to avoid explicit stack manipulations for bytecode operands. Rather, machine registers are used.

Assume we use the techniques of Section 15.3.1 to allocate registers “on the fly.” Explain how we could tag each bytecode, prior to code generation, with the machine registers the bytecode will use for its operands and result value. (These tags would then be used to “fill in” register names when bytecodes are expanded to machine code.)

5. A common subprogram optimization is inlining. At the point of a method call, the body of the method is substituted for the call, with actual parameter values used to initialize local variables that represent parameters.

Assume we have the bytecodes that represent the body of subprogram `P` that is marked `private` or `final` (and hence can't be redefined in a subclass). Assume further that `P` takes `n` parameters and uses `m` local variables. Explain how we could substitute the bytecodes representing `P`'s body for a call to `P`, prior to machine code generation. What changes in the body must be made to guarantee that the substituted bytecodes don't “clash” with other bytecodes in the context of call?

6. Show the expression tree, with `registerNeeds` labeling, that corresponds to the expression `a+(b+(c+((d+e)*(f/g))))`.

Show the code that would be generated using the `treeCG` code generator.

7. Recall that `registerNeeds` gives the *minimum* number of register needed to evaluate an expression without spilling registers to memory. Show that there exist expressions of *unbounded* size that require only 2 registers for evaluations. Show that for any value of `m` there exist expressions that always require *at least* `m` registers.
8. Some computer architectures include an immediate operation of the form


```
op $reg1,$reg2,val
```

 that computes `$reg1 = $reg2 op val`. In an immediate instruction `val` does not need to be loaded into a register; it is extracted directly from the instruction's bit pattern.

Explain how to extend `registerNeeds` and `treeCG` to accommodate architectures that include immediate operations.

9. Sometimes the code generated for an expression tree can be improved if the associative property of operators like `+` and `*` is exploited. For example, if the

following expression is translated using treeCG, four registers will be needed:
 $(a+b) * (c+d) * ((e+f) / (g-h))$

Even if the commutativity of $+$ and $*$ is exploited, four registers are still required. However, if the associativity of multiplication is exploited to evaluate multiplicands from right to left, then only three registers are needed. [First $((e+f) / (g-h))$ is evaluated, then $(c+d) * ((e+f) / (g-h))$, and finally $(a+b) * (c+d) * ((e+f) / (g-h))$.]

Write a routine `associate` that reorders the operands of associative operands to reduce register needs. (*Hint*: Allow associative operators to have more than two operands.)

10. In Section 15.4 we saw that many modern architectures are delayed load. That is, a value loaded into a register may not be used in the next instruction; a delay of one or more instructions is imposed (to allow time to access the cache).

The `treeCG` routine of Section 15.2 is not designed to handle delayed loads. Hence, it almost always generates instruction sequences that stall at selected loads.

Show that if an instruction sequence (of length 4 or more) generated by `treeCG` is given an additional register, it is possible to reorder the generated instructions to avoid *all* stalls for a processor with a one instruction load delay. (It will be necessary to reassign the register used by some operands to utilize the extra register).

11. Following the example of `doubleSum` in Section 15.3, convert the `stringSum` function of Section 15.1 into a form that makes explicit temporaries, live ranges, and parameter and return value register assignments. Then create the interference graph for `stringSum`. Use this interference graph and `GRegAlloc` to assign registers to `stringSum`, assuming three registers are available (including `$a0`, the parameter register and `$v0`, the return value register).
12. Assume we have the following method

```
int f(int i) {
    g(1, i);
}
```

At the point where the second parameter of `g` is loaded, we have a conflict if we require that parameters be passed in registers. In particular, `i` is passed in the first parameter register. But when the second parameter of `g` is loaded, the first parameter register is loaded with the value 1, possibly making `i` inaccessible. How can a register allocator deal with the problem of reuse of dedicated parameter registers? That is, what rules should be followed in determining where a parameter value is to be allocated throughout a program or subprogram?

13. In `GRegAlloc` we spill a live range if we are unable to color it. An alternative to spilling a live range is to split it, as is done in `PriorityRegAlloc`. What

changes are needed in `GRegAlloc` if we split an uncolorable live range rather than spill it?

14. At the site of a method call, we may need to save registers currently in use (lest they be overwritten by the method about to be executed). Assume we allocate registers using `GRegAlloc`. Explain how to determine which registers are in use at a method call.
15. Assume we have n registers available to allocate to a subprogram. Explain how, using either `GRegAlloc` or `PriorityRegAlloc`, we can estimate the total cost of register spills within the subprogram. How could this cost estimate be used in deciding how many registers to allocate to a subprogram?
16. In performing “on the fly” register allocation, some implementations store freed registers on a stack. Thus the most recently freed register will be the next register to be allocated. On the other hand, other implementations place freed registers at the back of a queue. Thus the least-recently freed register will be the next to be allocated.

From the point of view of a postpass code scheduler, which of the register reallocation implementations (stack vs. queue) is preferable? Why?

17. The `scheduleDag` code scheduler of Section 15.4 assumes that instructions that can stall have unit delay. That is, one instruction must separate an instruction that can stall from the first use of the value it produces. It may happen that some instructions have n cycle delays, meaning n instructions must separate the instruction from the first use of the value it produces.

How must `scheduleDag` be modified to handle instructions that have n cycle delays?

18. The `scheduleDag` code scheduler is a post-pass scheduler. That is, it schedules instructions *after* registers have been allocated. It is possible to create a dependency dag in terms of instructions that reference pseudo-registers. After instructions are scheduled, the pseudo-registers are mapped to real registers. Such a scheduler is a pre-pass scheduler, since it operates *before* register allocation.

It is important to note that the order in which instructions are scheduled will affect the number of registers that later are needed. For example, scheduling all loads immediately will force each load to use a different register. Scheduling some loads after other operations may allow registers to be reused.

If `scheduleDag` is used as a pre-pass code scheduler, how should it be modified so that the number of pseudo-registers in use is a criterion in selecting the next instructions to schedule? That is, scheduling an instruction that increases the number of registers that will be needed should be discouraged unless it serves to avoid stalls in the code schedule.

19. It is sometimes the case that we need to schedule a small block of code that forms the body of a frequently executed loop. For example

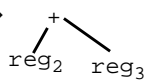
```
for (i=2; a <1000000; i++)
    a[i] = a[i-1]*a[i-2]/1000.0;
```

Operations like floating point multiplication and division often have significant delays (5 or more cycles). If a loop body is small, code scheduling can't do much—there aren't enough instructions to cover all the delays. In such a situation loop unrolling may help. The body of loop is replicated n times, with loop indices and loop limits suitably modified. For example, with $n = 2$, the above loop would become

```
for (i=2; a < 9999999; i+=2){
    a[i] = a[i-1]*a[i-2]/1000.0;
    a[i+1] = a[i]*a[i-1]/1000.0;}
```

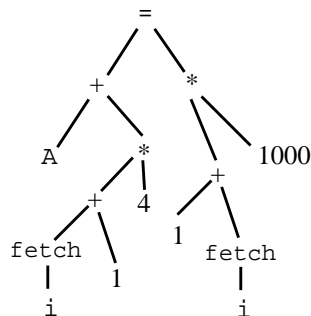
A larger loop body gives a code scheduler more instructions that can be placed after instructions that may stall. How can we determine the value of n (the loop unrolling factor) necessary to cover all (or most) of the delays in a loop body? What factors limit how large n (or an unrolled loop body) should be allowed to become?

20. The `scheduleDag` code scheduler is very optimistic with respect to loads—it schedules them assuming that they *always* hit in the primary cache. Real loads are not always so co-operative. Assume we can identify load instructions most likely to miss. How should `scheduleDag` be modified to use “probability of cache miss” information in scheduling instructions?
21. Assume we extend the IR tree patterns defined in Figure 15.15 with the following patterns for the MIPS add and load immediate instructions:

$reg_1 \rightarrow$

 $reg \rightarrow intlit$

`add $reg1, $reg2, $reg3 li $reg, intlit`

Show how the following IR tree, corresponding to $A[i+1] = (1+i) * 1000$, would be matched. What MIPS instructions would be generated?



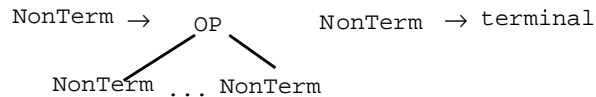
22. Code generators that use IR tree pattern matching still have the problem of allocating registers for the generated code. Suggest how an “on the fly” regis-

ter allocator can be integrated with pattern matching to form a complete code generator.

23. Instructions like the MIPS load immediate instruction are complicated by the fact that the immediate operand may be too big to fit in a single instruction. In fact immediate operands that are too big force two instructions to be generated—a “load upper immediate” that fills in the upper half of a word followed by an `or` immediate that fills in the lower half of a word.

How can costs and IR tree patterns be used to specify to an instruction selector that two alternative translations are possible depending on the *size* of an immediate operand?

24. Assume we have tree-structured instruction patterns limited to the two forms shown below



That is, a nonterminal may generate a single terminal symbol or it may generate an operator, all of whose children are nonterminals.

Give an algorithm that can walk any IR tree and determine whether it can be covered (matched) using a set of productions limited to the two forms described above.

25. Assume that we now add cost values (integer literals greater than or equal to 0) to instruction patterns limited to the two forms described in Exercise 24. Extend the algorithm you proposed in Exercise 24 so that it now finds a least-cost cover. That is, your algorithm should choose productions that minimize the overall cost of matching a given IR tree.
26. The following instruction sequence often appears in Java programs:

```
a[i] = ...
... = a[i];
```

That is, an element of an array is stored, then that same element is immediately reused. Suggest a peephole optimization rule, at the bytecode level, that would recognize this situation and optimize it using the `dup` bytecode.

27. Machines like the MIPS and SPARC have delayed branch instructions. That is, the instruction immediately following a branch is executed prior to transferring control to the target of the branch.

Often, compilers simply generate a `nop` instruction after a branch, effectively hiding the effects of the delayed branch. Suggest a peephole optimization pattern for unconditional branches followed by a `nop` that swaps the instruction prior to the branch into the “delay slot” that follows it. Can this optimization always be done, or must some conditions be met to make the swap valid?

Now consider a delayed conditional branch in which the value of a register is tested. If the condition is met, the instruction following the conditional branch is executed, and then the branch is taken. Otherwise, instructions following the conditional branch are executed (as usual); no branch is taken. Suggest a peephole optimization pattern that allows the instruction preceding a conditional branch to be moved after it as long as the swapped instruction does not affect the register tested by the conditional branch.

28. Many architectures include a load negative instruction that loads the negation of a value into a register. That is, the value, while being loaded, is subtracted from zero, with the difference stored into the register. Suggest a variety of instruction-level peephole optimization patterns that can make use of a load negative instruction.
29. After a peephole optimization is performed, the optimized instruction that is substituted for the original instructions is reconsidered for further peephole optimizations. Give examples of cases in which peephole optimizations may be profitably cascaded.
30. Assume we have a peephole optimizer that has n replacement patterns. The most obvious approach to implementing such an optimizer is to try each pattern in turn, leading to an optimizer whose speed is proportional to n .

Suggest an alternative implementation, based on hashing, that is largely independent of n . That is, the number of patterns considered may be doubled without automatically doubling the optimizer's execution time.

16

Program Optimization

This book has so far discussed the analysis and synthesis required to translate a programming language into interpretable or executable code. The analysis has been concerned with policing the programmer: making sure that the source program conforms to the definition of the programming language in which the program is written. After the compiler has verified that the source program conforms, synthesis takes over to translate the program. The target of this translation is typically an interpretable or executable instruction set. Thus, code generation consists of translating a portion of a program into a sequence of instructions that mean the same thing.

As is true of most languages, there are many ways to say the same thing. In Chapter Chapter:global:fifteen, *instruction selection* is presented as a mechanism for choosing an efficient sequence of instructions for the target machine. In this chapter, we examine more aggressive techniques for improving a program's performance. Section 16.1 introduces program optimization—its role in a compiler, its organization, and its potential for improving program performance. Section 16.2 presents **data flow analysis**—a technique for determining useful properties of a program at compile-time. Section 16.3 considers some advanced analysis and optimizations.

16.1 Introduction

When compilers were first pioneered, they were considered successful if programs written in a high-level language attained performance that rivaled hand-coded efforts. By today's standards, the programming languages of those days may seem primitive. However, the technology that achieved the requisite performance is very impressive—such techniques are successfully applied to modern programming languages. The scale of today's software projects would be impossible without the advent of advanced programming paradigms and languages. As a result, the goal of hand-coded performance has yielded to the goal of obtaining a *reasonable* fraction of a target machine's potential speed.

Meanwhile, the trend in **reduced instruction set computer** (RISC) architecture points toward comparatively low-level instruction sets. Such architectures feature shorter instruction times with correspondingly faster clock rates. Other developments include *liquid* architectures, whose operations, registers, and data paths can be reconfigured. Special-purpose instructions have also been introduced, such as the MMX instructions for Intel machines. These instructions facilitate graphics and numerical computations. Architectural innovations cannot succeed unless compilers can make effective use of both RISC and the more specialized instructions. Thus, collaboration between computer architects, language designers, and compiler writers continues to be strong.

The programming language community has defined the **semantic gap** as a (subjective) measure of the distance between a compiler's source and target languages. As this gap continues to widen, the compiler community is challenged to build efficient bridges. These challenges come from many sources—examples include object-orientation, mobile code, active network components, and distributed object systems. Compilers must produce excellent code quickly, quietly, and—of course—correctly.

16.1.1 Why Optimize?

Although its given name is a misnomer, it is certainly the goal of program optimization to improve a program's performance. Truly *optimal* performance cannot be achieved automatically, as the task subsumes problems that are known to be *undecidable* [?]. The four main areas in which program optimizers strive for improvement are as follows:

- High-level language features.
- Targeting of machine-specific features.
- The compiler itself.
- A better algorithm.

```

A ← B × C
function ×(Y, Z) : Matrix
  if Y.cols ≠ Z.rows 1
  then /* Throw an exception */
  else
    for i = 1 to Y.rows do
      for j = 1 to Z.cols do
        Result[i, j] ← 0
        for k = 1 to Y.cols do
          Result[i, j] ← Result[i, j] + Y[i, k] × Z[k, j]
    return (Result)
  end
procedure =(To, From) 2
  if To.cols ≠ From.cols or To.rows ≠ From.rows
  then /* Throw an exception */
  else
    for i = 1 to To.rows do
      for j = 1 to To.cols do
        To[i, j] ← From[i, j]
  end

```

Figure 16.1: Matrix multiplication using overloaded operators.

Each of these is considered next in some detail, using the example shown in Figure 16.1. In this program, variables A , B , and C are of type *Matrix*. For simplicity, we assume all matrices are of size $N \times N$. The \times and $=$ operators are overloaded to perform matrix multiplication and assignment, respectively, using the function and procedure provided in Figure 16.1.

High-Level Language Features

High-level languages contain features that offer flexibility and generality at the cost of some runtime efficiency. Optimizing compilers attempt to recover performance for such features in the following ways.

- Perhaps it can be shown that the feature is not used by some portion of a program.

In the example of Figure 16.1, suppose that the *Matrix* type is subtyped into *SymMatrix*—with definitions of the *Matrix* methods optimized for symmetric matrices. If A and B are actually of type *SymMatrix*, then languages that offer **virtual function dispatch** are obligated to call the most specialized method for an actual type. However, if compiler can show that \times and $=$ are not redefined in any subclass of *Matrix*, then the result of a virtual function dispatch on

```

for  $i = 1$  to  $N$  do                                     3
  for  $j = 1$  to  $N$  do
     $Result[i, j] \leftarrow 0$ 
    for  $k = 1$  to  $N$  do
       $Result[i, j] \leftarrow Result[i, j] + B[i, k] \times C[k, j]$       4
for  $i = 1$  to  $N$  do                                     5
  for  $j = 1$  to  $N$  do
     $A[i, j] \leftarrow Result[i, j]$ 

```

Figure 16.2: Inlining the overloaded operators.

these methods can be predicted for any object whose compile-time type is at least *Matrix*.

Based on such analysis, **method inlining** expands the method definitions in Figure 16.1 at their call sites, substituting the appropriate parameter values. Shown in Figure 16.2, the resulting program avoids the overhead of function calls. Moreover, the code is now specially tailored for its arguments, whose row and column sizes are N . Program optimization can then eliminate the tests at Steps 1 and 2.

- Perhaps it can be shown that a language-mandated operation is not necessary. For example, Java insists on subscript checks for array references and type-checks for **narrowing casts**. Such checks are unnecessary if an optimizing compiler can determine the outcome of the result at compile-time. When code is generated for Figure 16.2, **induction variable analysis** can show that i , j , and k all stay within the declared range of matrices A , B , and C . Thus, subscript tests can be eliminated for those arrays.

Even if the outcome is uncertain at compile-time, a test can be eliminated if its outcome is already computed. Suppose the compiler is required to check the subscript expressions for the *Result* matrix at Step 4 in Figure 16.2. Most likely, the code generator would insert a test for each reference of $Result[i, j]$. An optimization pass could determine that the second test is redundant.

Modern software-construction practices dictate that large software systems should be comprised of small, easily written, readily reusable components. As a result, the size of a compiler's **compilation unit**—the text directly presented in one run of the compiler—has been steadily dwindling. Optimizing compilers therefore consider the problem of **whole-program optimization** (WPO), which requires analyzing the interaction of a program's compilation units. Method inlining—successful in our example—is one example of the benefits of WPO. Even if a method cannot be inlined, WPO can generate a version of the invoked method that is customized to its calling context. In other words, the trend toward reusable code can result in systems that are general but not as efficient as they could be. Optimizing compilers step in to regain some of the lost performance.


```

for  $i = 1$  to  $N$  do
  for  $j = 1$  to  $N$  do
     $A[i, j] \leftarrow 0$ 
    for  $k = 1$  to  $N$  do 6
       $A[i, j] \leftarrow A[i, j] + B[i, k] \times C[k, j]$  7

```

Figure 16.3: Fusing the loop nests.

Target-Specific Optimization

Portability ranks high among the goals of today's programming languages. Ideally, once a program is written in a high-level language, it should be movable without modification to any computing system that supports the language. Architected interpretive languages such as **Java virtual machine** (JVM) support portability nicely—any computer that sports a JVM interpreter can run any Java program. But the question remains—how fast? Although most modern computers are based on RISC principles, the details of their instruction sets vary widely. Moreover, various models for a given architecture can also differ greatly with regard to their storage hierarchies, their instruction timings, and their degree of concurrency.

Continuing with our example, the program shown in Figure 16.2 is an improvement over the version shown in Figure 16.1, but it is possible to obtain better performance. Consider the behavior of the matrix *Result*. The values of *Result* are computed in the loop nest at Step 3—one element at a time. Each of these elements is then copied from *Result* to *A* by the loop nest at Step 5. Poor performance can be expected on any **non-uniform memory access** (NUMA) system that cannot accommodate *Result* at its fastest storage level. Better performance can be obtained if the data is stored directly in *A*. Optimizing compilers that feature **loop fusion** can identify that the outer two loop nests at Steps 3 and 5 are structurally equivalent. **Dependence analysis** can show that each element of *Result* is computed independently. The loops can then be fused to obtain the program shown in Figure 16.3, in which the *Result* matrix is eliminated.

Artifacts of program translation

In the process of translating a program from a source to target language, a compiler pass can introduce spurious computations. As discussed in Chapter Chapter:global:fifteen, compilers try to keep frequently accessed variables in fast registers. Thus, it is likely that the iteration variables in Figure 16.3 are kept in registers. Figure 16.4 shows the results of straightforward code generation for the loops in Figure 16.3.

- The loops contain instructions at Steps 10, 11, and 12 that save the iteration variable register in the named variable. However, this particular program does not require a value for the iteration variables when the loops are finished. Thus, such saves are unnecessary. In Chapter Chapter:global:fifteen,

```

ri ← 1
while ri ≤ N do
  rj ← 1
  while rj ≤ N do
    rA ← * (Addr(A) + (((ri - 1) × N + (rj - 1))) × 4)
    *(ra) ← 0
    rk ← 1
    while rk ≤ N do
      rA ← * (Addr(A) + (((ri - 1) × N + (rj - 1))) × 4)           8
      rB ← * (Addr(B) + (((ri - 1) × N + (rk - 1))) × 4)
      rC ← * (Addr(C) + (((rk - 1) × N + (rj - 1))) × 4)
      rsum ← rA
      rprod ← rB × rC
      rsum ← rsum + rprod
      rA ← * (Addr(A) + (((i - 1) × N + (j - 1))) × 4)           9
      *(ra) ← rsum
      rk ← rk + 1
    k ← rk                               10
    rj ← rj + 1                           11
  j ← rj
  ri ← ri + 1                               12
i ← ri

```

Figure 16.4: Low-level code sequence.

register allocation can avoid such saves if the iteration variable can be allocated a register without spilling.

- Because code generation is mechanically invoked for each program element, it is easy to generate redundant computations. For example, Steps 8 and 9 compute the address of $A[i, j]$. Only one such computation is necessary.

Conceivably, the code generator could be written to account for these conditions as it generates code. Superficially, this may seem desirable, but modern compiler design practices dictate otherwise.

- Such concerns can greatly complicate the task of writing the code generator. Issues such as instruction selection and register allocation are the primary concerns of the code generator. Generally, it is wiser to craft a compiler by combining simple, single-purpose transformations that are easily understood, programmed, and maintained. Each such transformation is often called a **pass** of the compiler.
- There are typically many portions of a compiler that can generate superfluous code. It is more efficient to write one compiler pass that is dedicated to the

removal of unnecessary computations than to duplicate that functionality throughout the compiler.

In this chapter, we study two program optimizations that remove unnecessary code: **dead-code elimination** and **unreachable code elimination**. Even the most basic compiler typically includes these passes, if only to clean up after itself.

Continuing with our example, consider code generation for Figure 16.3. Let us assume that each array element occupies 4 bytes, and that subscripts are specified in the range of $1..N$. The code for indexing the array element $A[i, j]$ becomes

$$\text{Addr}(A) + (((i - 1) \times N + (j - 1))) \times 4$$

which takes

4	integer “+” and “-”
2	integer “×”
6	integer operations

Since Step 7 has 4 such array references, each execution of this statement takes

16	integer “+” and “-”
8	integer “×”
3	loads
1	floating “+”
1	floating “×”
1	store
30	instructions

Moreover, the loop contains 2 floating-point instructions and 24 fixed-point instructions. On superscalar architectures that support concurrent fixed- and floating-point instructions, this loop can pose a severe bottleneck for the fixed-point unit.

The computation can be greatly improved by optimizations that are described in this chapter.

- **Loop-invariant detection** can determine that the (address) expression $A[i, j]$ does not change at Step 7.
- **Reduction in strength** can replace the address computations for the matrices with simple increments of an index variable. The iteration variables themselves can disappear, with loop termination based on the subscript addresses.

The result of applying these optimizations on the innermost loop is shown in Figure 16.5. The inner loop now contains 2 floating-point and 2 fixed-point operations—a balance that greatly improves the loop's performance on modern processors.

```

FourN ← 4 × N
for i = 1 to N do
  for j = 1 to N do
    a ← &(A[i, j])
    b ← &(B[i, 1])
    c ← &(C[1, j])
    while b < &(B[i, 1]) + FourN do
      *a ← *a + *b × *c
      b ← b + 4
      c ← c + FourN

```

Figure 16.5: Optimized matrix multiply.

Program Restructuring

Some optimizing compilers attempt to improve a program by prescribing better algorithms and data structures.

- Some programming languages contain constructs for which diverse implementations are possible. For example, Pascal and SETL offer the *set* type-constructor. An optimizing compiler could choose a particular implementation for a set, based on the the predicted construction and use of the set.
- If the example in Figure 16.3 can be recognized as matrix-multiply, then an optimizing compiler could replace the code with a better algorithm for matrix-multiply. Except in the most idiomatic of circumstances, it is difficult for a compiler to recognize an algorithm at work.

16.1.2 Organization

In this section, we briefly examine the organization of an optimizing compiler—how it represents and processes programs.

Use of an intermediate language

The task of creating an optimizing compiler can be substantial, especially given the extent of automation available for a compiler's other parts. Ideally, the design, cost, and benefits of an optimizing compiler can be shared by multiple source languages and target architectures. An **intermediate language** (IL) such as JVM can serve as a focal point for the optimizer, as shown in Figure 16.6. The optimizer operates on the IL until the point of code generation, when target code is finally generated. Attaining this level of machine-independent optimization is difficult, as each target machine and source language can possess features that demand special treatment. Nonetheless, most compiler designs resemble the structure shown in Figure 16.6.

The techniques we present in this chapter are sufficiently general to accommodate any program representation, given that the following information is available.

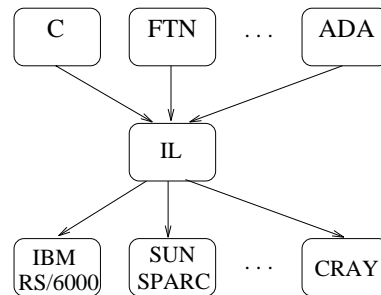


Figure 16.6: Program optimization for multiple sources and targets.

- The operations of interest must be known and explicitly represented. For example, if the program contains a code sequence that computes $a + b$, the compiler might also be called upon to perform this computation.
- The effects of the program's operations must be known or approximated. For example, the optimizer may require knowing the set of global variables that can be affected by a given method. If this set is not known, it can be approximated by the set of all global variables.
- The potential ordering of operations may be of interest. Within a given method, its operations can usually be regarded as the nodes of a **control flow graph**, whose edges show the potential flow from one operation to the next. Any path taken by a program at runtime is a path of this control flow graph. Of course, the graph may contain paths that are rarely taken, but these are necessary to understand the program's potential behavior. Moreover, the graph can contain paths that can never be taken—consider a predicate whose outcome is always false.

As discussed in Section 16.2, each optimization pass has its own view of the important aspects of the program's operations. We stress here that the potential run-time behavior of a program is approximated at compile-time, both in terms of the operations' effects and order.

A series of small passes

As discussed in Section 16.1.1, it is expedient to organize an optimizing compiler as a series of *passes*. Each pass should have a narrowly defined goal, so that its scope is easily understood. This facilitates development of each pass as well as the integration of the separate passes into an optimizing compiler. For example, it should be possible to run the *dead-code elimination* pass at any point to remove useless code. Certainly this pass would be run prior to code generation. However, the effectiveness of an individual pass as well as the overall speed of the optimizing compiler can be improved by sweeping away unnecessary code.

To facilitate development and interoperability, it is useful for each pass to accept and produce programs in the IL of the compiler. Thus, the program transformations shown in Figures 16.1, 16.2, 16.3, and 16.5 could be carried out on the compiler's intermediate form. In research compilers, a source language (such as C) can serve as a nice IL, because a C compiler can take the optimized IL form to native code when the optimizing compiler is finished. Alternatively JVM can serve as an IL for high-level program optimization, although JVM is certainly biased toward a Java view of what programs can do.

Unfortunately, the passes of a compiler can interact in subtle ways. For example, **code motion** can rearrange a program's computations to make better use of a given architecture. However, as the distance between a variable's definition and its uses grows, so does the pressure on the register allocator. Thus, it is often the case that a compiler's optimizations are at odds with each other. Ideally, the passes of an optimizing compiler should be fairly independent, so that they can be reordered and retried as situations require.

```

u ← 5
repeat
  if r
  then
    v ← 9
    if p
    then u ← 6
    else w ← 5
    x ← v + w
  else y ← v + w
  u ← 7
  repeat
    if q
    then
      z ← v + w  13
  until r
v ← 2
until s
    
```

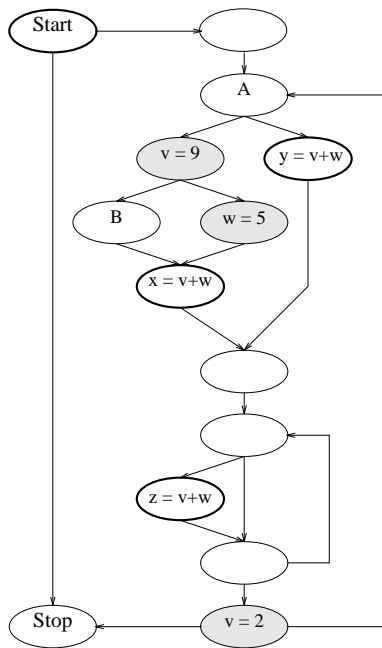


Figure 16.7: A program and its control flow graph.

16.2 Data Flow Analysis

As discussed in Section 16.1, an optimizing compiler is typically organized as a series of **passes**. Each pass may require approximate information about the program's run-time behavior. **Data flow frameworks** offer a structure for such analysis. We begin in Section 16.2.1 by describing several **data flow problems**—their specification and their use in program optimization. In Section 16.2.2, we formalize the notion of a data flow framework. Section 16.2.3 describes how to evaluate a data flow framework. In this section, we use the program shown in Figure 16.7(a) as an example. The same program is represented in Figure 16.7(b) by its **control flow graph**—the instructions are contained in the nodes, and the edges indicate potential branching within the program.

16.2.1 Introduction and Examples

In Section 16.2.2, we offer a more formal presentation of data flow frameworks. Here, we discuss data flow problems informally, examining a few popular optimization problems and reasoning about their data flow formulation. In each problem, we are interested in the following.

- What is the effect of a code sequence on the solution to the problem?

- When branching in a program converges, how do we summarize the solution so that we need not keep track of branch-specific behavior?
- What are the best and worst possible solutions?

Local answers to the above questions are combined by data flow analysis to arrive at a global solution.

Available Expressions

Figure 16.7 contains several computations of the expression $v + w$. If we can show that the particular value of $v + w$ computed at Step **13** is already *available*, then there is no need to recompute the expression at Step **13**. More specifically, an expression $expr$ is **available** at edge e of a flow graph if the past behavior of the program necessarily includes a computation of the value of $expr$ at edge e . The **available expressions** data flow problem analyzes programs to determine such information.

To solve the **available expressions** problem, a compiler must examine a program to determine that expression $expr$ is available at edge e , regardless of how the program arrives at edge e . Returning to our example, $v + w$ is **available** at Step **13** if every path arriving at Step **13** computes $v + w$ without a subsequent change to v or w .

In this problem, an instruction affects the solution if it computes $v + w$ or if it changes the value of v or w , as follows.

- The Start node of the program is assumed to contain an implicit computation of $v + w$. For programs that initialize their variables, $v + w$ is certainly available after node Start. Otherwise, although $v + w$ is uninitialized, the compiler is free to assume the expression has *any* value it chooses.
- A node of the flow graph that computes $v + w$ makes $v + w$ available.
- A node of the flow graph that assigns v or w makes $v + w$ not available.
- All other nodes have no effect on the availability of $v + w$.

In Figure 16.7(b), the shaded nodes make $v + w$ unavailable. The nodes with dark circles make $v + w$ available. From the definition of this problem, we summarize two solutions by assuming the worst case. At the input to node A, the path from Start contains an implicit computation of $v + w$; on the loop edge, $v + w$ is not available. At the input to node A, we must therefore assume that $v + w$ is *not* available.

Based on the above reasoning, information can be pushed through the graph to reach the solution shown on each edge of Figure 16.8. In particular, $v + w$ is available on the edge entering the node that represents Step **13** in Figure 16.7(a). Thus, the program can be optimized by eliminating the recomputation of $v + w$. Similarly, the address computation for $A[i, j]$ need not be performed at Step **9** in Figure 16.4—the expression is available from the computation at Step **8**.

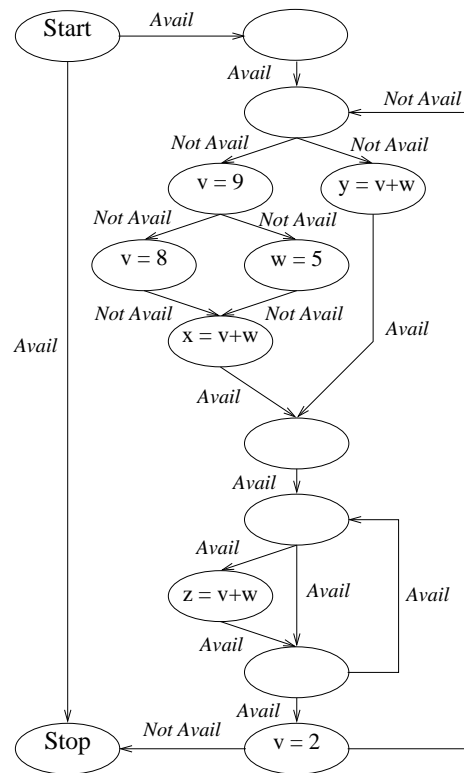


Figure 16.8: Global solution for availability of the expression $v + w$.

In this example, we explored the availability of a single expression $v + w$. Better optimization could obviously result from determining that an expression is available. In an optimizing compiler, one of the following situations usually holds.

- The compiler identifies an expression such as $v + w$ as *important*, in the sense that eliminating its computation can significantly improve the program's performance. In this situation, the optimizing compiler may selectively evaluate the availability of a single expression.
- The compiler may compute availability of *all* expressions, without regard to the importance of the results. In this situation, it is common to formulate a *set* of expressions and compute the availability of its members. Section 16.2.2 and Exercise 8 considers this in greater detail.

Live Variables

We next examine an optimization problem related to register allocation. As discussed in Chapter Chapter:global:fifteen, k registers suffice for a program whose

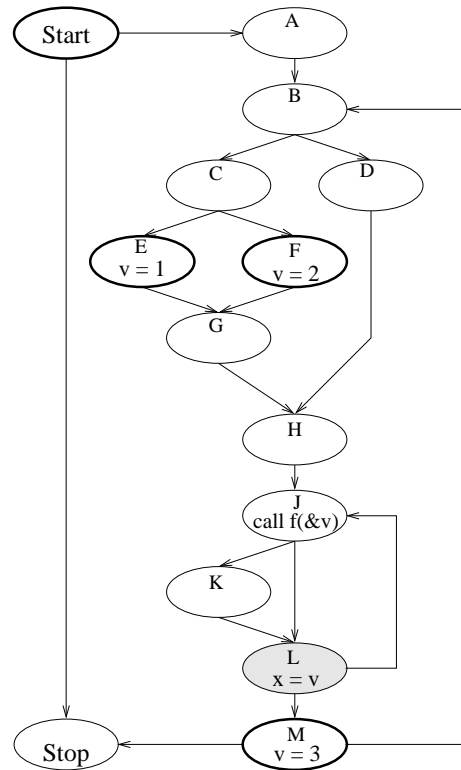
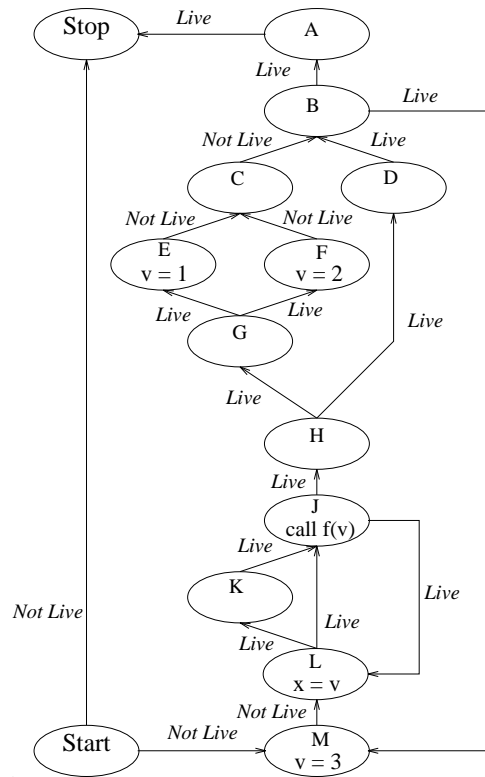


Figure 16.9: Example flow graph for liveness. The function f potentially assigns v but does not read its value.

interference graph is k -colorable. In this graph, each node represents one of the program's variables. An edge is placed between two nodes if their associated variables are simultaneously *live*. A variable v is *live* at control flow graph edge e if the future behavior of the program can reference the value of v that is present at edge e . In other words, the value of a live variable is potentially of future use in the program. Thus, register allocation relies on **live variable analysis** to build the interference graph.

In the available-expressions problem, information was pushed forward through the control flow graph. In the live-variables problem, the potential behavior of a program is pushed backward through the control flow graph. In the control flow graph shown in Figure 16.9, consider the liveness of variable v . The dark-circled nodes contain uses of v —they represent future behavior that makes v live. On the other hand, the shaded nodes destroy the current value of v . Such nodes represent future behavior that makes v not live. At the `Stop` node, we may assume v is dead since the program is over. If the value of v were to survive an execution, then its

Figure 16.10: Solution for liveness of v .

value should be printed by the program.

Figure 16.9 shows a node with a `call` instruction. How does this node affect the liveness of v ? In this example, we assume that the function f potentially assigns v but does not use its value. Thus, the invoked function does not make v live. However, since we cannot be certain that f assigns v , the invoked function does not make v dead. This particular node has no effect on the solution to live variables.

Based on the definition of this problem, common points of control flow cause v to be live if *any* future behavior causes v to be live. The solution for liveness of v is shown in Figure 16.10—the control flow edges are reversed to show how the computation is performed. It is clearly to an optimizing compiler's advantage to show that a variable is *not* live. Any resources associated with a dead variable can be reclaimed by the compiler, including the variable's register or local JVM slot.

An optimizing compiler may seek liveness information for one variable or for a set of variables. Exercise 9 considers the computation for a set.

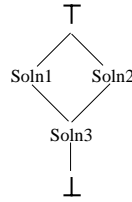


Figure 16.11: A meet lattice.

16.2.2 Formal Specification

We have introduced the notion of a data flow framework informally, relying on examples drawn from optimizing compilers. In this section, we formalize the notion of a **data flow framework**. As we examine these details, it will be helpful to refer to the problems discussed in Section 16.2.1.

A data flow framework has the following components.

- A **flow graph**. This directed graph's nodes typically represent some aspect of a program's behavior. For example, a node may represent a nonbranching sequence of instructions, or an entire procedure. The graph's edges represent a relation over the nodes. For example, the edges may indicate potential transfer of control by branching or by procedure call. We assume the graph's edges are oriented in the “direction” of the data flow problem.
- A **meet lattice**. This is a mathematical structure that describes the solution space of the data flow problem and designates how to combine multiple solutions in a safe (conservative) way. It is convenient to present such lattices as Hasse diagrams: to find the meet of two elements, you put one finger on each element, and travel down the page until your fingers first meet. For example, the lattice in Figure 16.11 indicates that when *Soln1* and *Soln2* must be combined, then their solutions are to be approximated by *Soln3*.
- A set of **transfer functions**. These model the behavior of a node with respect to the optimization problem under study. Figure 16.12 depicts a generic transfer function. A transfer function's input is the solution that holds on entry to the node, so that the function's output specifies how the node behaves given the input solution.

We next examine each of these components in detail.

Data Flow Graph

The **data flow graph** is constructed for an optimization problem, so that evaluation of this graph produces a solution to the problem.

- Transfer functions are associated with each node;

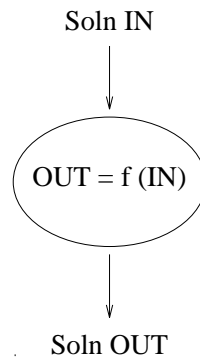


Figure 16.12: A node's transfer function.

- Information converging at a node is combined as directed by the meet lattice;
- Information is propagated through the data flow graph to obtain a solution.

For the problems considered here, a flow graph's nodes represent some component of a program's behavior and its edges represent potential transfer of control between nodes. In posing an optimization problem as a data flow framework, the resulting framework is said to be

- **forward**, if the solution at a node can depend only on the program's past behavior. Evaluating such problems involves propagating information *forward* through the flow graph. Thus, the control flow graph in Figure 16.7(b) served as the data flow graph for availability analysis of $v + w$ in the program of Figure 16.7.
- **backward**, if the solution at a node can depend only on the program's future behavior. The live variables problem introduced in Section 16.2.1 is such a problem. For live variables, a suitable data flow graph is the reverse control flow graph, as shown in Figure 16.10.
- **bidirectional**, if both past and future behavior is relevant.

In this chapter, we discuss only forward or backward problems; moreover, we assume that edges in the data flow graph are oriented in the direction of the data flow problem. With this assumption, information always propagates in the direction of the data flow graph's edges. It is convenient to augment data flow graphs with a Start and Stop node, and an edge from Start to Stop, as shown in Figure 16.7(b).

In compilers where space is at a premium, nodes of a control flow graph typically correspond to the *maximal* straight-line sequences—the **basic blocks**—of a program. While this design conserves space, program analysis and optimization must then occur at two levels: *within* and *between* the basic blocks. These two levels are called **local** and **global** data flow analysis, respectively. An extra level of

analysis complicates and increases the expense of writing, maintaining, and documenting an optimizing compiler. We therefore formulate data flow graphs whose nodes model the effects of perhaps a single JVM or MIPS instruction. In a production compiler, the choice of node composition may dictate otherwise.

Meet Lattice

As with all lattices, the **meet lattice** represents a partial order imposed on a set. Formally, the meet lattice is defined by the tuple

$$(A, \top, \perp, \preceq, \wedge)$$

which has the following components.

- A **solution space** A . In a data flow framework, the relevant set is the space of all possible solutions to the data flow problem. Exercise 9 considers the live-variables problem, posed over a set of n variables. Since each variable is either live or not live, the set of possible solutions contains 2^n elements. Fortunately, we need not enumerate or represent all elements in this large set. In fact, some data flow problems have an infinite solution space.
- The **meet operator** \wedge . The partial order present in the lattice directs how to combine (summarize) multiple solutions to a data flow problem. In Figure 16.8, the first node of the outer loop receives two edges— $v+w$ is available on one edge and not available on the other. The meet operation (\wedge) serves to summarize the two solutions. Mathematically, \wedge is associative, so multiple solutions are easily summarized by applying \wedge pairwise in any order.
- Distinguished elements \top and \perp . Lattices associated with data flow frameworks always include the following distinguished elements of A .
 - \top intuitively represents the solution that allows the most optimization.
 - \perp intuitively represents the solution that prevents or inhibits optimization.
- The comparison operator \preceq . The meet lattice includes a reflexive partial order, denoted by \preceq . Given two solutions a and b —both from set A —it must be true that that $a \preceq b$ or $a \not\preceq b$. If $a \preceq b$, then solution a is no better than solution b . Further, if $a \prec b$, then solution a is strictly worse than b —optimization based on solution a will not be as good as with solution b . If $a \not\preceq b$, then solutions a and b are incomparable.

For example, consider the problem of live variables, computed for the set of variables $\{v, w\}$. As discussed in Section 16.2.1, the storage associated with variables found *not* to be live can be reused. Thus, optimization is improved when fewer variables are found to be live. Thus, the set $\{v, w\}$ is worse than the set $\{v\}$ or the set $\{w\}$. However, the solution $\{v\}$ cannot be compared with the set $\{w\}$. In both cases, one variable is live, and data flow analysis cannot prefer one to the other for live variables.

Property	Explanation
$a \wedge a = a$	The combination of two identical solutions is trivial.
$a \preceq b \iff a \wedge b = a$	If a is worse than b , then combining a and b must yield a ; if $a = b$, then the combination is simply a , as above.
$a \wedge b \preceq a$ $a \wedge b \preceq b$	The combination of a and b can be no better than a or b .
$a \wedge \top = a$	Since \top is the best solution, combining with \top changes nothing.
$a \wedge \perp = \perp$	Since \perp is the worst solution, any combination that includes \perp will be \perp .

Figure 16.13: Meet lattice properties.

At this point, it is important to develop an intuitive understanding of the lattice—especially its distinguished elements \top and \perp . For each analysis problem, there is some solution that admits the greatest amount of optimization. This solution is always \top in the lattice. Recalling available expressions, the best solution would make *every* expression available—all recomputations could then be eliminated. Correspondingly, \perp represents the solution that admits the least amount of optimization. For available expressions, \perp implies that *no* expressions can be eliminated.

Mathematically, the meet lattice has the properties shown in Figure 16.13.

Transfer Functions

Finally, our data flow framework needs a mechanism to describe the effects of a fragment of program code—specifically, the code represented by a path through the flow graph. Consider a single node of the data flow graph. A solution is present on entry to the node, and the transfer function is responsible for converting this input solution to a value that holds after the node executes. Mathematically, the node's behavior can be modeled as a *function* whose domain and range are the meet lattice A . This function must be **total**—defined on every possible input. Moreover, we shall require the function to behave **monotonically**—the function cannot produce a better result when presented with a worse input. Consider the available expressions problem, posed for the expressions $v+w$ and $a+b$. Figure 16.14 contains fragments of a program and explains the transfer function that models their effects.

Fragment	Transfer Function	Explanation
$y = v+w$	$f(in) = in \cup \{ "v+w" \}$	Regardless of which expressions were available on entry to this node, expression “v+w” becomes available after the node. That status of expression $a + b$ is not affected by this node.
$v = 9$	$f(in) = in - \{ "v+w" \}$	Regardless of which expressions were available on entry to this node, expression “v+w” is <i>not</i> available after the node, because the assignment to v potentially changes the value of “v+w”, and the node includes no recomputation of this expression. The status of expression $a + b$ is not affected.
<code>printf("hello")</code>	$f(in) = in$	This node affects no expression; thus, the solution on exit from the node is identical to the solution on entry.

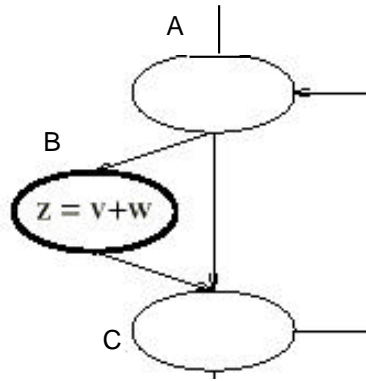
Figure 16.14: Data flow transfer functions.

Because a transfer functions are *mathematical*, they can model not only the effects of a single node but also the effects along a *path* through a program. If a node with transfer function f is followed by a node with transfer function g , then the cumulative effect of both nodes on input a is captured by $g(f(a))$. In other words, program behavior—brief or lengthy—is captured by a transfer function. A transfer function can be applied to any lattice value a to obtain the compile-time estimation of the program fragment's behavior given the conditions represented by a .

16.2.3 Evaluation **WARNING this subsection is incomplete**

Having discussed how to pose an optimization problem in a data flow framework, we next turn to evaluating such frameworks. As shown in Figure 16.12, each node of the data flow graph asserts a transfer function. Once we know the input value to a transfer function, its output value is easily computed. When multiple solutions converge at a node, the meet lattice serves to summarize the solutions, as shown in Figure 16.11. Thus, we can compute the input to a node as the meet of all the outputs that feed the node.

Figure 16.15: Iterative evaluation of a data flow framework.

Figure 16.16: Is $v + w$ available throughout this loop?

An algorithm for evaluating a data flow framework is shown in Figure 16.15.

Two issues: initialization – is it OK to do it to top? and termination – how do we know that the loop terminates?

Initialization

The problem is that the solution at a given node depends on the solution at other nodes, as dictated by the edges of the data flow graph. In fact, the solution at a node can depend directly or indirectly *on itself*. Figure 16.7 contains a loop that is shown in Figure 16.16. The solution in Figure 16.8 shows that $v + w$ is available everywhere inside the loop. But how is this determined? The nodes in Figure 16.16 have the following transfer functions.

$$\begin{aligned} f_A(in_A) &= in_A \\ f_B(in_B) &= \top \\ f_C(in_C) &= in_C \end{aligned}$$

Let in_{loop} denote the input to the loop—the input to node A from outside the loop. The inputs to A and C are computed as follows.

$$\begin{aligned} in_C &= f_B(in_B) \wedge f_A(in_A) \\ &= \top \wedge f_A(in_A) \\ &= f_A(in_A) \\ in_A &= f_C(in_C) \wedge in_{loop} \\ &= in_C \wedge in_{loop} \\ &= f_A(in_A) \wedge in_{loop} \end{aligned}$$

In other words, the input to node A depends on the output of node A ! This seems a contradiction, unless we reason that the first time we evaluate A 's transfer function, we can assume some prior result. Computationally, we have the following choices concerning the prior result.

$$\begin{aligned} f_A(in_A) &= \perp \\ f_A(in_A) &= \top \end{aligned}$$

It seems safe to assume that $v + w$ is *not* available previously—that is, $f_A(in_A) = \perp$. Based on this assumption, $v + w$ is not available anywhere except in node B . It is more daring to assume that $v + w$ *is* available— $f_A(in_A) = \top$. Based on this assumption, we obtain the result shown in Figure 16.8, with $v + w$ available everywhere in the inner loop.

Termination

In a **monotone data flow framework**, each transfer function f obeys the rule

$$a \preceq b \Rightarrow f(a) \preceq f(b)$$

for all pairs of lattice values a and b . Note that if the lattice does not relate a and b , the rule is trivially satisfied.

16.2.4 Application of Data Flow Frameworks

We have thus far studied the formulation and evaluation of data flow frameworks. We examine next a series of optimization problems and discuss their solution as data flow problems.

Available Expressions

Section 16.2.1 contained an informal description of the available expressions problem for the expression $v + w$. Suppose a compiler is interested in computing the availability of a *set* of expressions, where the elements of the set are chosen by examining the expressions of a program. For example, a program with the quadratic formula

$$-b + \sqrt{\frac{b^2 - 4ac}{2a}}$$

contains the variables a , b , and c . Each variable is itself a simple expression. However, each variable is trivially available by loading its value. The more ambitious expressions in this formula include $-b$, b^2 , ac , $4ac$, $b^2 - 4ac$. We might prefer to show that $b^2 - 4ac$ is available, because this expression is costly to compute. However, the availability of any of these expressions can improve the program's performance.

We can generalize the available expressions problem to accommodate a *set* of expressions, as follows.

- This is a forward problem. The data flow graph is simply the control flow graph.
- If S is the set of interesting expressions, then the solution space A for this problem is 2^S —the **power set** of S . In other words, a given expression is or is not present in any particular solution.
- The most favorable solution is S —every expression is available. Thus, $\top = S$ and $\perp = \emptyset$.
- The transfer function at node Y can be formulated as

$$f_Y(in) = (in - Kill_Y) \cup Gen_Y$$

where

- $Kill_Y$ is the set of expressions containing a variable that is (potentially) modified by node Y .
- Gen_Y is the set of expressions computed by node Y .

If the flow graph's nodes are sufficiently small—say, a few instructions—then it is possible to limit each node's effects to obtain $Kill_Y \cap Gen_Y = \emptyset$ at each node Y . In other words, none of these small nodes both generates an expression and kills it. Exercise 6 explores this issue in greater detail.

- The meet operation is accomplished by set intersection, based on the following reasoning.
 - An expression is available only if all paths reaching a node compute the expression.
 - Formally, we require that $\top \wedge a = a$. If \top is the set of all expressions, then meet must be set intersection.

As discussed in Exercise 7, some simplifications can be made when computing available expressions for a single expression. Available expressions is one of the so-called **bit-vectoring data flow problems**. Other such problems are described in Exercises 9, 10, and 11.

Constant Propagation

In this optimization, a compiler attempts to determine expressions whose value is constant over all executions of a program. Most programmers do not intentionally introduce constant expressions. Instead, such expressions arise as artifacts of program translation, as discussed in Section 16.1. Interprocedurally, constants develop when a general method is invoked with arguments that are constant.

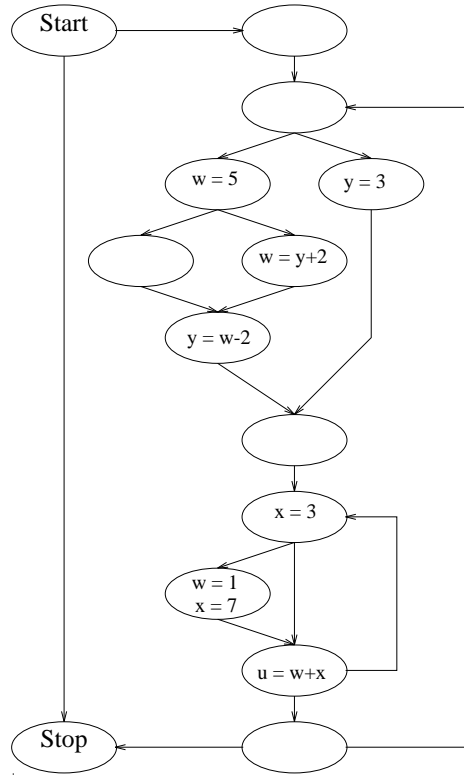


Figure 16.17: A program for constant propagation.

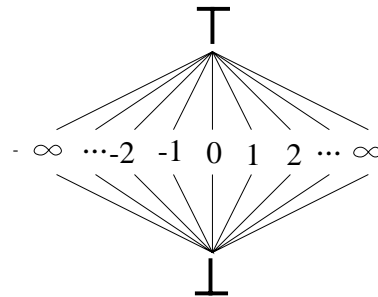


Figure 16.18: Lattice for constant propagation---one expression only.

Consider the program whose control flow graph is shown in Figure 16.17. Some nodes assign constant values to variables. In other nodes, these constant values may combine to create other constant values. We can describe constant propagation as a data flow problem as follows.

- Without loss of generality, we pose constant propagation over a program's variables. If the program contains an expression or subexpression of interest, this can be assigned to a temporary variable. Constant propagation can then try to discover a constant value for the temporary.
- For a single variable, we formulate the three-tiered lattice shown in Figure 16.18.
 - \top means that the variable is considered a constant of some (as yet) undetermined value.
 - \perp means that the expression is not constant.
 - Otherwise, the expression has a constant value found in the middle layer.
- As shown in Figure 16.19, each edge in the data flow graph carries a solution for each variable of interest.
- The meet operator is applied using the lattice shown in Figure 16.18. The lattice is applied separately for each variable of interest.
- The transfer function at node Y interprets the node by substituting the node's incoming solution for each variable used in the node's expression. Suppose node v is computed using the variables in set U . The solution for variable v after node Y is computed as follows.
 - If any variable in U has value \perp , then v has value \perp .
 - Otherwise, if any variable in U has value \top , then v has value \top .
 - Otherwise, all variables in U have constant value. The expression is evaluated and the constant value is assigned to v .

Figure 16.19¹ shows the solution for constant propagation on the program of Figure 16.17.

This program demonstrates some interesting properties of constant propagation. Although x is uninitialized prior to executing the node that assigns its value, constant propagation is free to assume it has the value 3 throughout the program. This occurs because an uninitialized variable has the solution \top . When summarized by the meet operator, $\top \wedge 3 = 3$. Although the uninitialized variable may indicate a programming error, the optimizer can assume an uninitialized variable has any value it likes without fear of contradiction. If the programming language at hand has semantics that insist on initialization of all variables—say to 0—then this must be represented by an assignment to these variables at the Start node.

¹*Production note: s*

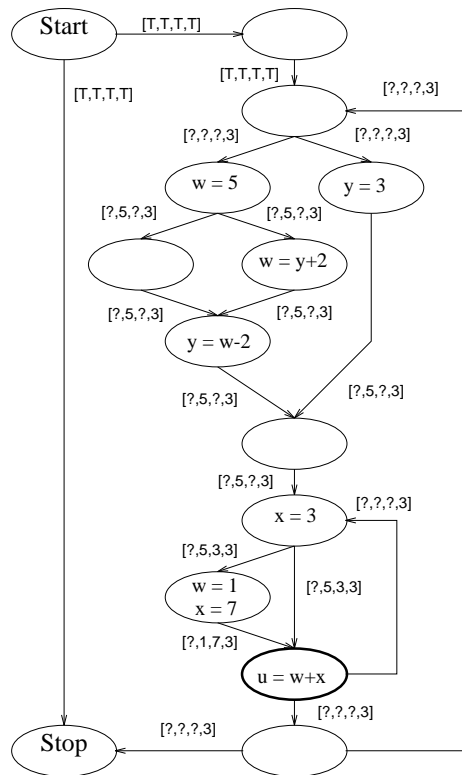


Figure 16.19: Constant propagation. Each edge is labeled by a tuple that shows the solution for $[u, w, x, y]$.

Another observation is that constant propagation failed to determine that u has the value 8 at the dark-circled node in Figure 16.19. Prior to this node, both w and x have constant values. Although their sum is the same, the individual values for these variables differ. Thus, when a meet is taken at the dark-circled node, w and x both appear to be \perp . For efficiency, data flow analysis computes its solution based on edges and not paths into a node. For some data flow problems, this approach provides the best solution possible—such is the case for the bit-vectoring data flow problems discussed in Exercises 12 and 18. Unfortunately, the example in Figures 16.17 and 16.19 shows that constant propagation does not compute the best solution.

16.3 Advanced Optimizations

16.3.1 SSA Form

Definition

Construction

16.3.2 SSA-based Transformations

Constant Propagation

Value Numbering

Code Motion

16.3.3 Loop Transformations

Should we even go here?

Summary

hello

Exercises

1. Recall the transformation experienced by the inner loops as the program in Figure 16.1 was optimized into the program shown in Figure 16.5. Apply these same transformations to the outer loops of Figure 16.5.
2. In Section 16.1.2, JVM is proposed as an intermediate form. Compare and contrast your rendering of the program shown in Figure 16.5 in the following intermediate forms.
 - (a) The C programming language
 - (b) JVM
 - (c) The MIPS instruction set
3. Compile—automatically or by-hand—the program shown in Figure 16.5 to generate the intermediate forms of Exercise 2. Compare the sizes of the representations.
4. Consider each of the following properties of a proposed **intermediate language** (IL). Explain how each property is or is not found in each IL of Exercise 2.
 - (a) The IL should be a *bona fide* language—it should have a precise syntactic and semantic specification. It should not exist simply as an aggregation of data structures.
 - (b) The size of programs expressed in the IL should be as compact as possible.
 - (c) The IL should have an expanded, human-readable form.
 - (d) The IL should be sufficiently general to represent the important aspects of a wide variety of source languages.
 - (e) The IL should be easily and cleanly extensible.
 - (f) The IL should be sufficiently general to support generation of efficient code for multiple target architectures.
5. Using a common programming language, construct a program whose control flow graph is the one shown in Figure 16.9.
6. Section 16.2.4 describes how to apply data flow frameworks. The section includes the following formulation for transfer functions in the data flow problem of available expressions.

$$f_Y(in) = (in - Kill_Y) \cup Gen_Y$$

where

- $Kill_Y$ is the set of expressions containing a variable that is (potentially) modified by node Y .

- Gen_Y is the set of expressions computed by node Y .

Suppose a flow graph node represents the following code.

```

 $v \leftarrow z$ 
 $u \leftarrow v + w$ 
 $v \leftarrow x$ 

```

For the expression $v + w$, the node kills the expression, generates it, and then kills it again. The cumulative effect is to kill the expression, but this requires analyzing the order in which the operations occur inside the node. Describe how to formulate a data flow graph with *smaller* nodes so that the order of operations within the nodes need not be examined.

- Section 16.2.4 presents the formal definition of a data flow framework to determine the availability of expressions in the set S . Describe the simplifications that result when computing available expressions for a single expression—when $|S| = 1$.
 - What is the lattice?
 - What are \top and \perp ?
 - What are the transfer functions?
 - How is meet performed?
- The **bit-vectoring data flow problems** earn their name from a common representation for finite sets—the **bit vector**. In this representation, a slot is reserved for each element in the set. If $e \in S$, then e 's slot is **true** in the bit vector that represents set S .

Describe how bit vectors can be applied to the available expressions problem for a set of n expressions. In particular, describe how a bit vector is affected by

- the transfer function at a node
 - application of the meet operator
 - assignment to \top or \perp
- Explain how liveness of a set of n variables can be computed as a data flow problem.
 - Define the formal framework, using the notation in Section 16.2.2. The transfer function at node Y is defined by the following formula (from Section 16.2.4).

$$f_Y(in) = (in - Kill_Y) \cup Gen_Y$$

Explain how $Kill_Y$ and Gen_Y are determined for live variables at node Y .

- (b) Now consider the use of bit vectors to solve live variables. The transfer function can be implemented as described in Exercise 8. How is the meet operation performed? What are \top and \perp ?
10. Liveness shows that a variable is potentially of future use in a program. The **very busy expressions** problem if an expression's value is *certainly* of future use.
- Is this a forward or backward problem?
 - What is the best solution?
 - Describe the effects of a node on an expression.
 - How are solutions summarized at common control flow points?
 - How would you determine liveness for a *set* of expressions?
11. Reaching defs
12. Four of the data flow problems presented in Section 16.2 and in Exercises 10 and 11 are:
- Available expressions
 - Live variables
 - Very busy expressions
 - Reaching definitions

These problems are known as the **bit-vectoring data flow problems**. Summarize these problems by entering each into its proper position in the following table.

	Forward	Backward
Any		
All		

The columns refer to whether information is pushed forward or backward to achieve a solution to the problem. The rows refer to whether information should hold on all paths or any path.

13. A data flow framework is **monotone** if the following formula holds for every transfer function f and lattice values a and b .

$$x \preceq y \Rightarrow f(x) \preceq f(y).$$

In other words, a transfer function cannot produce a better answer given a worse input. Prove that the following formula must hold for monotone frameworks.

$$f(a \wedge b) \preceq f(a) \wedge f(b).$$

14. A data flow framework is **rapid** if it defines `rapid`. Then prove that available expressions is rapid.
15. Generalize the proof from Exercise 14 to prove or disprove that all four bit-vectoring data flow problems in Exercise 12 are rapid.
16. Prove or disprove that constant propagation is a rapid data flow problem.
17. A data flow problem is **distributive** if the following formula holds for every transfer function f and lattice values a and b .

$$f(a \wedge b) = f(a) \wedge f(b).$$

Prove or disprove that available expressions (Exercise 8) is a distributive data flow problem.

18. Generalize the proof from Exercise 17 to prove or disprove that all four bit-vectoring data flow problems in Exercise 12 are distributive.
19. Prove or disprove that constant propagation is a distributive data flow problem.
20. Consider generalizing the problem of constant propagation to that of *range analysis*. For each variable, we wish to associate a minimum and maximum value, such that the actual value of the variable (at that site in the program) at runtime is guaranteed to fall between the two values. For example, consider the following program.

```

x ← 5
y ← 3
if p
then
    z ← x + y           14
else
    z ← x - y           15
w ← z

```

After their assignment, variable x has range $5 \dots 5$ and variable y has range $3 \dots 3$. The effect of Step **14** gives z the range $8 \dots 8$. The effect of Step **15** gives z the range $2 \dots 2$. The assignment for w therefore gets the range $2 \dots 8$.

- (a) Sketch the data flow lattice for a single variable. Be specific about the values for \top and \perp .
 - (b) Is this a forwards or backwards propagation problem?
 - (c) If the variable v could have range r_1 or r_2 , describe how to compute the meet of these two ranges.
21. As defined in Exercise 20, prove or disprove that range analysis is a rapid data flow problem.

22. As defined in Exercise 20, prove or disprove that range analysis is a distribute data flow problem.
23. The number of bits problem
 - (a) Prove or disprove that this data flow problem is distribute.
 - (b) Prove or disprove that this data flow problem is rapid.