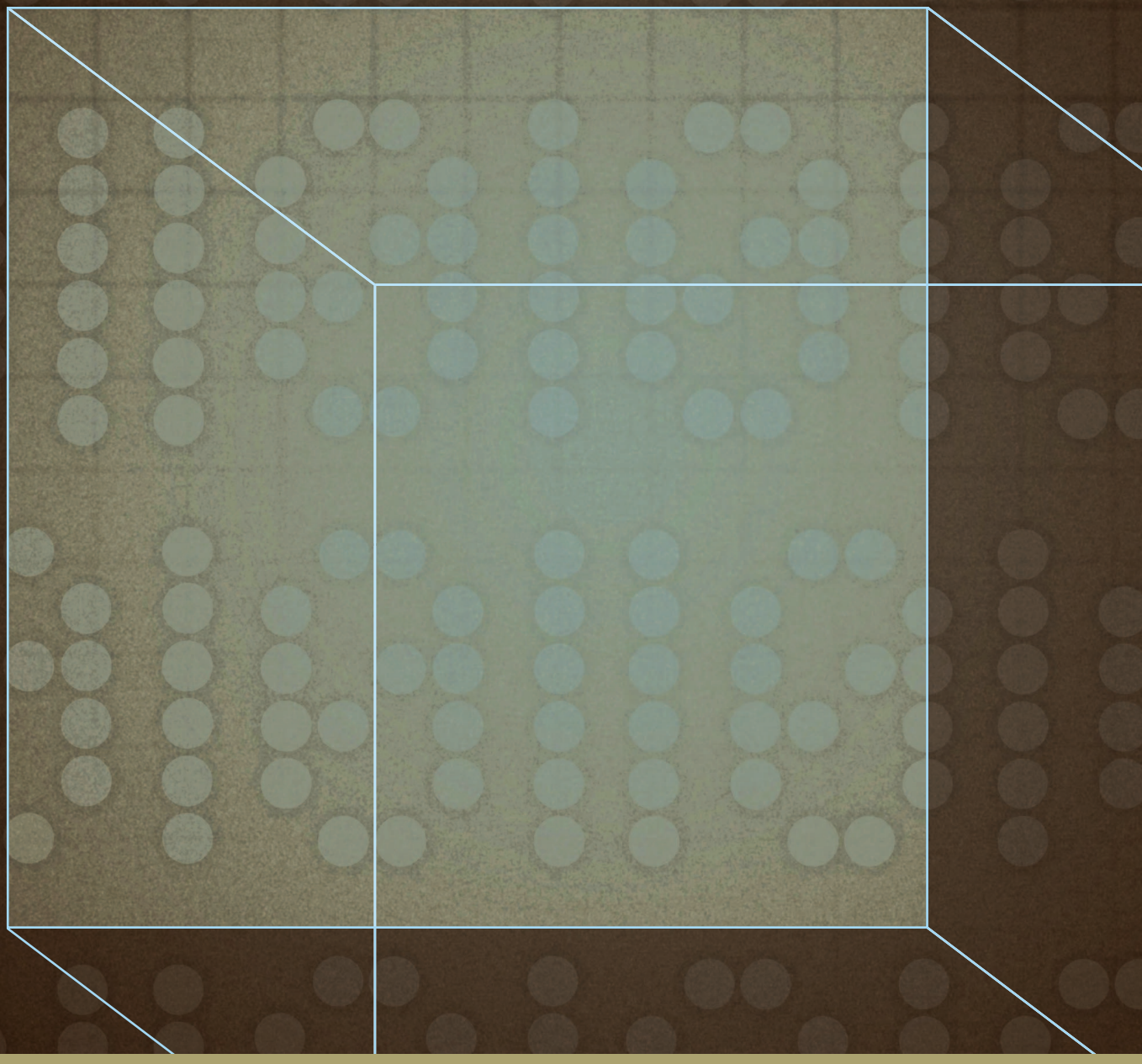
PEARSON NEW INTERNATIONAL EDITION

# Compilers
# Principles, Techniques, and Tools

Aho Lam Sethi Ullman

Second Edition



ALWAYS LEARNING

PEARSON

# Pearson New International Edition

Compilers
Principles, Techniques, and Tools
Aho Lam Sethi Ullman
Second Edition

PEARSON

**Pearson Education Limited**
Edinburgh Gate
Harlow
Essex CM20 2JE
England and Associated Companies throughout the world

*Visit us on the World Wide Web at:* www.pearsoned.co.uk




ISBN 10: 1-292-02434-8
ISBN 13: 978-1-292-02434-9

**British Library Cataloguing-in-Publication Data**
A catalogue record for this book is available from the British Library

Printed in the United States of America

# Table of Contents

# Chapter 1

# Introduction

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.

The software systems that do this translation are called *compilers*.

This book is about how to design and implement compilers. We shall discover that a few basic ideas can be used to construct translators for a wide variety of languages and machines. Besides compilers, the principles and techniques for compiler design are applicable to so many other domains that they are likely to be reused many times in the career of a computer scientist. The study of compiler writing touches upon programming languages, machine architecture, language theory, algorithms, and software engineering.

In this preliminary chapter, we introduce the different forms of language translators, give a high level overview of the structure of a typical compiler, and discuss the trends in programming languages and machine architecture that are shaping compilers. We include some observations on the relationship between compiler design and computer-science theory and an outline of the applications of compiler technology that go beyond compilation. We end with a brief outline of key programming-language concepts that will be needed for our study of compilers.

## 1.1 Language Processors

Simply stated, a compiler is a program that can read a program in one language — the *source* language — and translate it into an equivalent program in another language — the *target* language; see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

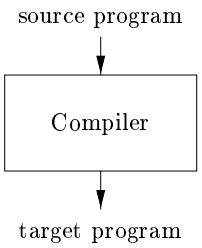



Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.

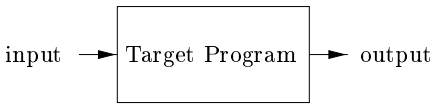



Figure 1.2: Running the target program

An *interpreter* is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

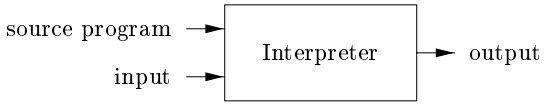



Figure 1.3: An interpreter

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

**Example 1.1 :** Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called *bytecodes*. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

In order to achieve faster processing of inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input. □

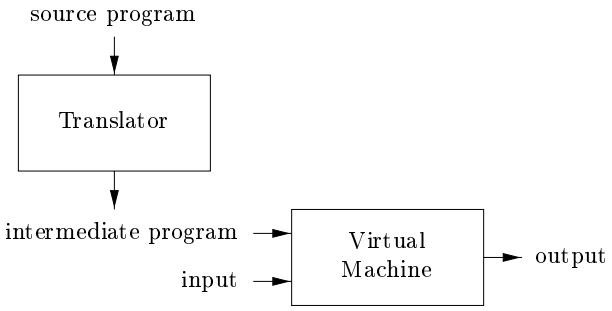



Figure 1.4: A hybrid compiler

In addition to a compiler, several other programs may be required to create an executable target program, as shown in Fig. 1.5. A source program may be divided into modules stored in separate files. The task of collecting the source program is sometimes entrusted to a separate program, called a *preprocessor*. The preprocessor may also expand shorthands, called macros, into source language statements.

The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output.

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together all of the executable object files into memory for execution.

### 1.1.1 Exercises for Section 1.1

**Exercise 1.1.1:** What is the difference between a compiler and an interpreter?

**Exercise 1.1.2:** What are the advantages of (a) a compiler over an interpreter (b) an interpreter over a compiler?

**Exercise 1.1.3:** What advantages are there to a language-processing system in which the compiler produces assembly language rather than machine language?

**Exercise 1.1.4:** A compiler that translates a high-level language into another high-level language is called a *source-to-source* translator. What advantages are there to using C as a target language for a compiler?

**Exercise 1.1.5:** Describe some of the tasks that an assembler needs to perform.

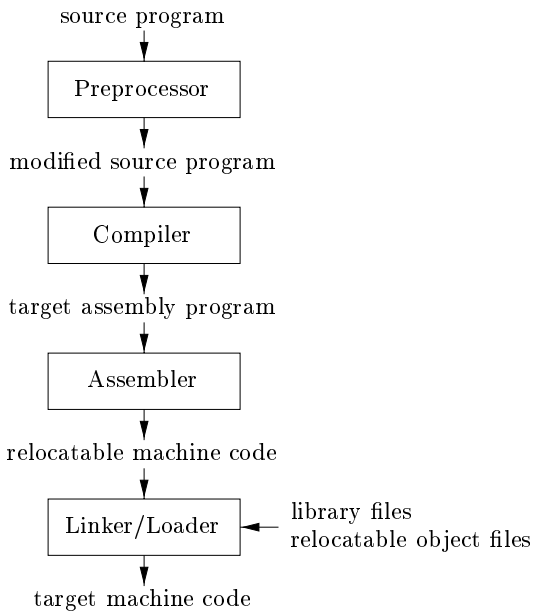



Figure 1.5: A language-processing system

## 1.2 The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.

The *analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a *symbol table*, which is passed along with the intermediate representation to the synthesis part.

The *synthesis* part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the *front end* of the compiler; the synthesis part is the *back end*.

If we examine the compilation process in more detail, we see that it operates as a sequence of *phases*, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig. 1.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the

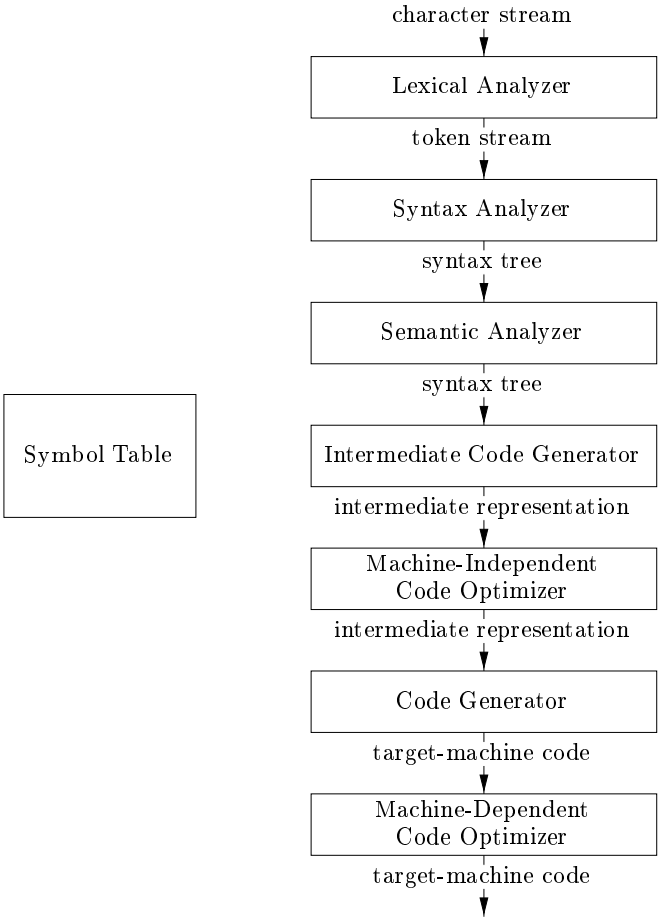



Figure 1.6: Phases of a compiler

entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the front end and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation. Since optimization is optional, one or the other of the two optimization phases shown in Fig. 1.6 may be missing.

### 1.2.1 Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program

and groups the characters into meaningful sequences called *lexemes*. For each lexeme, the lexical analyzer produces as output a *token* of the form

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

that it passes on to the subsequent phase, syntax analysis. In the token, the first component *token-name* is an abstract symbol that is used during syntax analysis, and the second component *attribute-value* points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation.

For example, suppose a source program contains the assignment statement

```
position = initial + rate * 60
```
(1.1)

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. `position` is a lexeme that would be mapped into a token $\langle \mathbf{id}, 1 \rangle$, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol-table entry for `position`. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol `=` is a lexeme that is mapped into the token $\langle = \rangle$. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. `initial` is a lexeme that is mapped into the token $\langle \mathbf{id}, 2 \rangle$, where 2 points to the symbol-table entry for `initial`.

4. `+` is a lexeme that is mapped into the token $\langle + \rangle$.

5. `rate` is a lexeme that is mapped into the token $\langle \mathbf{id}, 3 \rangle$, where 3 points to the symbol-table entry for `rate`.

6. `*` is a lexeme that is mapped into the token $\langle * \rangle$.

7. `60` is a lexeme that is mapped into the token $\langle 60 \rangle$.[1]

Blanks separating the lexemes would be discarded by the lexical analyzer.

Figure 1.7 shows the representation of the assignment statement (1.1) after lexical analysis as the sequence of tokens

$$\langle \mathbf{id}, 1 \rangle \ \langle = \rangle \ \langle \mathbf{id}, 2 \rangle \ \langle + \rangle \ \langle \mathbf{id}, 3 \rangle \ \langle * \rangle \ \langle 60 \rangle \tag{1.2}$$

In this representation, the token names $=$, $+$, and $*$ are abstract symbols for the assignment, addition, and multiplication operators, respectively.

[1] Technically speaking, for the lexeme `60` we should make up a token like $\langle \mathbf{number}, 4 \rangle$, where 4 points to the symbol table for the internal representation of integer 60 but we shall defer the discussion of tokens for numbers until Chapter 2. Chapter 3 discusses techniques for building lexical analyzers.

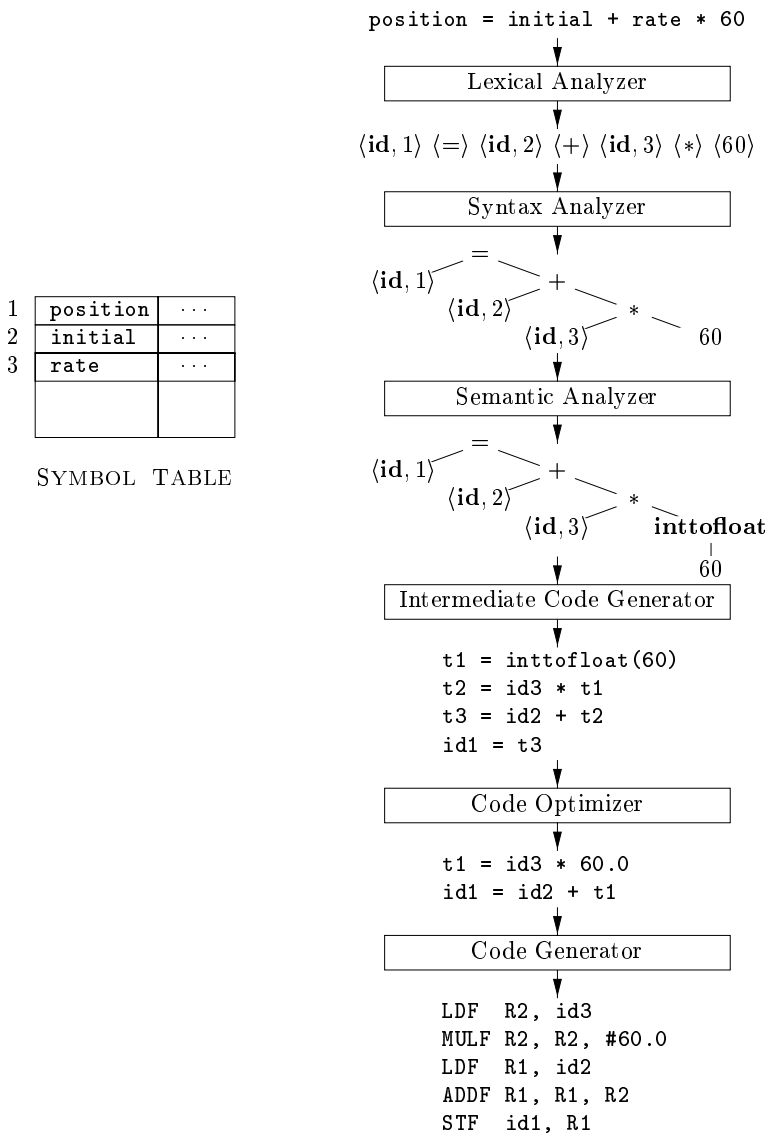



Figure 1.7: Translation of an assignment statement

### 1.2.2 Syntax Analysis

The second phase of the compiler is *syntax analysis* or *parsing*. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a *syntax tree* in which each interior node represents an operation and the children of the node represent the arguments of the operation. A syntax tree for the token stream (1.2) is shown as the output of the syntactic analyzer in Fig. 1.7.

This tree shows the order in which the operations in the assignment

```
position = initial + rate * 60
```

are to be performed. The tree has an interior node labeled $*$ with $\langle \mathbf{id}, 3 \rangle$ as its left child and the integer 60 as its right child. The node $\langle \mathbf{id}, 3 \rangle$ represents the identifier `rate`. The node labeled $*$ makes it explicit that we must first multiply the value of `rate` by 60. The node labeled $+$ indicates that we must add the result of this multiplication to the value of `initial`. The root of the tree, labeled $=$, indicates that we must store the result of this addition into the location for the identifier `position`. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

The subsequent phases of the compiler use the grammatical structure to help analyze the source program and generate the target program. In Chapter 4 we shall use context-free grammars to specify the grammatical structure of programming languages and discuss algorithms for constructing efficient syntax analyzers automatically from certain classes of grammars. In Chapters 2 and 5 we shall see that syntax-directed definitions can help specify the translation of programming language constructs.

### 1.2.3 Semantic Analysis

The *semantic analyzer* uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is *type checking*, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

The language specification may permit some type conversions called *coercions*. For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers. If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.

Such a coercion appears in Fig. 1.7. Suppose that `position`, `initial`, and `rate` have been declared to be floating-point numbers, and that the lexeme `60` by itself forms an integer. The type checker in the semantic analyzer in Fig. 1.7 discovers that the operator $*$ is applied to a floating-point number `rate` and an integer 60. In this case, the integer may be converted into a floating-point number. In Fig. 1.7, notice that the output of the semantic analyzer has an extra node for the operator **inttofloat**, which explicitly converts its integer argument into a floating-point number. Type checking and semantic analysis are discussed in Chapter 6.

### 1.2.4 Intermediate Code Generation

In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

In Chapter 6, we consider an intermediate form called *three-address code*, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register. The output of the intermediate code generator in Fig. 1.7 consists of the three-address code sequence

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```
(1.3)

There are several points worth noting about three-address instructions. First, each three-address assignment instruction has at most one operator on the right side. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program (1.1). Second, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some "three-address instructions" like the first and last in the sequence (1.3), above, have fewer than three operands.

In Chapter 6, we cover the principal intermediate representations used in compilers. Chapter 5 introduces techniques for syntax-directed translation that are applied in Chapter 6 to type checking and intermediate-code generation for typical programming language constructs such as expressions, flow-of-control constructs, and procedure calls.

### 1.2.5 Code Optimization

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power. For example, a straightforward algorithm generates the intermediate code (1.3), using an instruction for each operator in the tree representation that comes from the semantic analyzer.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time, so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0. Moreover, `t3` is used only once to transmit its value to `id1` so the optimizer can transform (1.3) into the shorter sequence

```
t1 = id3 * 60.0
id1 = id2 + t1
```
(1.4)

There is a great variation in the amount of code optimization different compilers perform. In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much. The chapters from 8 on discuss machine-independent and machine-dependent optimizations in detail.

### 1.2.6 Code Generation

The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

For example, using registers `R1` and `R2`, the intermediate code in (1.4) might get translated into the machine code

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```
(1.5)

The first operand of each instruction specifies a destination. The `F` in each instruction tells us that it deals with floating-point numbers. The code in

(1.5) loads the contents of address `id3` into register `R2`, then multiplies it with floating-point constant 60.0. The `#` signifies that 60.0 is to be treated as an immediate constant. The third instruction moves `id2` into register `R1` and the fourth adds to it the value previously computed in register `R2`. Finally, the value in register `R1` is stored into the address of `id1`, so the code correctly implements the assignment statement (1.1). Chapter 8 covers code generation.

This discussion of code generation has ignored the important issue of storage allocation for the identifiers in the source program. As we shall see in Chapter 7, the organization of storage at run-time depends on the language being compiled. Storage-allocation decisions are made either during intermediate code generation or during code generation.

### 1.2.7 Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly. Symbol tables are discussed in Chapter 2.

### 1.2.8 The Grouping of Phases into Passes

The discussion of phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a *pass* that reads an input file and writes an output file. For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.

Some compiler collections have been created around carefully designed intermediate representations that allow the front end for a particular language to interface with the back end for a certain target machine. With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for that target machine. Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

### 1.2.9 Compiler-Construction Tools

The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on. In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.

These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms. The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler. Some commonly used compiler-construction tools include

1. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming language.

2. *Scanner generators* that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. *Syntax-directed translation engines* that produce collections of routines for walking a parse tree and generating intermediate code.

4. *Code-generator generators* that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.

6. *Compiler-construction toolkits* that provide an integrated set of routines for constructing various phases of a compiler.

We shall describe many of these tools throughout this book.

## 1.3 The Evolution of Programming Languages

The first electronic computers appeared in the 1940's and were programmed in machine language by sequences of 0's and 1's that explicitly told the computer what operations to execute and in what order. The operations themselves were very low level: move data from one location to another, add the contents of two registers, compare two values, and so on. Needless to say, this kind of programming was slow, tedious, and error prone. And once written, the programs were hard to understand and modify.

### 1.3.1 The Move to Higher-level Languages

The first step towards more people-friendly programming languages was the development of mnemonic assembly languages in the early 1950's. Initially, the instructions in an assembly language were just mnemonic representations of machine instructions. Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

A major step towards higher-level languages was made in the latter half of the 1950's with the development of Fortran for scientific computation, Cobol for business data processing, and Lisp for symbolic computation. The philosophy behind these languages was to create higher-level notations with which programmers could more easily write numerical computations, business applications, and symbolic programs. These languages were so successful that they are still in use today.

In the following decades, many more languages were created with innovative features to help make programming easier, more natural, and more robust. Later in this chapter, we shall discuss some key features that are common to many modern programming languages.

Today, there are thousands of programming languages. They can be classified in a variety of ways. One classification is by generation. *First-generation languages* are the machine languages, *second-generation* the assembly languages, and *third-generation* the higher-level languages like Fortran, Cobol, Lisp, C, C++, C#, and Java. *Fourth-generation languages* are languages designed for specific applications like NOMAD for report generation, SQL for database queries, and Postscript for text formatting. The term *fifth-generation language* has been applied to logic- and constraint-based languages like Prolog and OPS5.

Another classification of languages uses the term *imperative* for languages in which a program specifies *how* a computation is to be done and *declarative* for languages in which a program specifies *what* computation is to be done. Languages such as C, C++, C#, and Java are imperative languages. In imperative languages there is a notion of program state and statements that change the state. Functional languages such as ML and Haskell and constraint logic languages such as Prolog are often considered to be declarative languages.

The term *von Neumann language* is applied to programming languages whose computational model is based on the von Neumann computer architecture. Many of today's languages, such as Fortran and C are von Neumann languages.

An *object-oriented language* is one that supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another. Simula 67 and Smalltalk are the earliest major object-oriented languages. Languages such as C++, C#, Java, and Ruby are more recent object-oriented languages.

*Scripting languages* are interpreted languages with high-level operators designed for "gluing together" computations. These computations were originally

called "scripts." Awk, JavaScript, Perl, PHP, Python, Ruby, and Tcl are popular examples of scripting languages. Programs written in scripting languages are often much shorter than equivalent programs written in languages like C.

### 1.3.2 Impacts on Compilers

Since the design of programming languages and compilers are intimately related, the advances in programming languages placed new demands on compiler writers. They had to devise algorithms and representations to translate and support the new language features. Since the 1940's, computer architecture has evolved as well. Not only did the compiler writers have to track new language features, they also had to devise translation algorithms that would take maximal advantage of the new hardware capabilities.

Compilers can help promote the use of high-level languages by minimizing the execution overhead of the programs written in these languages. Compilers are also critical in making high-performance computer architectures effective on users' applications. In fact, the performance of a computer system is so dependent on compiler technology that compilers are used as a tool in evaluating architectural concepts before a computer is built.

Compiler writing is challenging. A compiler by itself is a large program. Moreover, many modern language-processing systems handle several source languages and target machines within the same framework; that is, they serve as collections of compilers, possibly consisting of millions of lines of code. Consequently, good software-engineering techniques are essential for creating and evolving modern language processors.

A compiler must translate correctly the potentially infinite set of programs that could be written in the source language. The problem of generating the optimal target code from a source program is undecidable in general; thus, compiler writers must evaluate tradeoffs about what problems to tackle and what heuristics to use to approach the problem of generating efficient code.

A study of compilers is also a study of how theory meets practice, as we shall see in Section 1.4.

The purpose of this text is to teach the methodology and fundamental ideas used in compiler design. It is not the intention of this text to teach all the algorithms and techniques that could be used for building a state-of-the-art language-processing system. However, readers of this text will acquire the basic knowledge and understanding to learn how to build a compiler relatively easily.

### 1.3.3 Exercises for Section 1.3

**Exercise 1.3.1:** Indicate which of the following terms:

| | | |
|---|---|---|
| a) imperative | b) declarative | c) von Neumann |
| d) object-oriented | e) functional | f) third-generation |
| g) fourth-generation | h) scripting | |

apply to which of the following languages:

| | | | | |
|---|---|---|---|---|
| 1) C | 2) C++ | 3) Cobol | 4) Fortran | 5) Java |
| 6) Lisp | 7) ML | 8) Perl | 9) Python | 10) VB. |

## 1.4 The Science of Building a Compiler

Compiler design is full of beautiful examples where complicated real-world problems are solved by abstracting the essence of the problem mathematically. These serve as excellent illustrations of how abstractions can be used to solve problems: take a problem, formulate a mathematical abstraction that captures the key characteristics, and solve it using mathematical techniques. The problem formulation must be grounded in a solid understanding of the characteristics of computer programs, and the solution must be validated and refined empirically.

A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

### 1.4.1 Modeling in Compiler Design and Implementation

The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms, while balancing the need for generality and power against simplicity and efficiency.

Some of most fundamental models are finite-state machines and regular expressions, which we shall meet in Chapter 3. These models are useful for describing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also among the most fundamental models are context-free grammars, used to describe the syntactic structure of programming languages such as the nesting of parentheses or control constructs. We shall study grammars in Chapter 4. Similarly, trees are an important model for representing the structure of programs and their translation into object code, as we shall see in Chapter 5.

### 1.4.2 The Science of Code Optimization

The term "optimization" in compiler design refers to the attempts that a compiler makes to produce code that is more efficient than the obvious code. "Optimization" is thus a misnomer, since there is no way that the code produced by a compiler can be guaranteed to be as fast or faster than any other code that performs the same task.

In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively parallel computers require substantial optimization, or their performance suffers by orders of magnitude. With the likely prevalence of multicore machines (computers with chips that have large numbers of processors on them), all compilers will have to face the problem of taking advantage of multiprocessor machines.

It is hard, if not impossible, to build a robust compiler out of "hacks." Thus, an extensive and useful theory has been built up around the problem of optimizing code. The use of a rigorous mathematical foundation allows us to show that an optimization is correct and that it produces the desirable effect for all possible inputs. We shall see, starting in Chapter 9, how models such as graphs, matrices, and linear programs are necessary if the compiler is to produce well optimized code.

On the other hand, pure theory alone is insufficient. Like many real-world problems, there are no perfect answers. In fact, most of the questions that we ask in compiler optimization are undecidable. One of the most important skills in compiler design is the ability to formulate the right problem to solve. We need a good understanding of the behavior of programs to start with and thorough experimentation and evaluation to validate our intuitions.

Compiler optimizations must meet the following design objectives:

- The optimization must be correct, that is, preserve the meaning of the compiled program,
- The optimization must improve the performance of many programs,
- The compilation time must be kept reasonable, and
- The engineering effort required must be manageable.

It is impossible to overemphasize the importance of correctness. It is trivial to write a compiler that generates fast code if the generated code need not be correct! Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free! Thus, the most important objective in writing a compiler is that it is correct.

The second goal is that the compiler must be effective in improving the performance of many input programs. Normally, performance means the speed of the program execution. Especially in embedded applications, we may also wish to minimize the size of the generated code. And in the case of mobile devices, it is also desirable that the code minimizes power consumption. Typically, the same optimizations that speed up execution time also conserve power. Besides performance, usability aspects such as error reporting and debugging are also important.

Third, we need to keep the compilation time short to support a rapid development and debugging cycle. This requirement has become easier to meet as

machines get faster. Often, a program is first developed and debugged without program optimizations. Not only is the compilation time reduced, but more importantly, unoptimized programs are easier to debug, because the optimizations introduced by a compiler often obscure the relationship between the source code and the object code. Turning on optimizations in the compiler sometimes exposes new problems in the source program; thus testing must again be performed on the optimized code. The need for additional testing sometimes deters the use of optimizations in applications, especially if their performance is not critical.

Finally, a compiler is a complex system; we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable. There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems. The approach used in compiler development involves both theory and experimentation. We normally start by formulating the problem based on our intuitions on what the important issues are.

## 1.5 Applications of Compiler Technology

Compiler design is not only about compilers, and many people use the technology learned by studying compilers in school, yet have never, strictly speaking, written (even part of) a compiler for a major programming language. Compiler technology has other important uses as well. Additionally, compiler design impacts several other areas of computer science. In this section, we review the most important interactions and applications of the technology.

### 1.5.1 Implementation of High-Level Programming Languages

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language. Generally, higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code. Unfortunately, lower-level programs are harder to write and — worse still — less portable, more prone to errors, and harder to maintain. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions.

**Example 1.2:** The **register** keyword in the C programming language is an early example of the interaction between compiler technology and language evolution. When the C language was created in the mid 1970s, it was considered necessary to let a programmer control which program variables reside in registers. This control became unnecessary as effective register-allocation techniques were developed, and most modern programs no longer use this language feature.

In fact, programs that use the **register** keyword may lose efficiency, because programmers often are not the best judge of very low-level matters like register allocation. The optimal choice of register allocation depends greatly on the specifics of a machine architecture. Hardwiring low-level resource-management decisions like register allocation may in fact hurt performance, especially if the program is run on machines other than the one for which it was written. □

The many shifts in the popular choice of programming languages have been in the direction of increased levels of abstraction. C was the predominant systems programming language of the 80's; many of the new projects started in the 90's chose C++; Java, introduced in 1995, gained popularity quickly in the late 90's. The new programming-language features introduced in each round spurred new research in compiler optimization. In the following, we give an overview on the main language features that have stimulated significant advances in compiler technology.

Practically all common programming languages, including C, Fortran and Cobol, support user-defined aggregate data types, such as arrays and structures, and high-level control flow, such as loops and procedure invocations. If we just take each high-level construct or data-access operation and translate it directly to machine code, the result would be very inefficient. A body of compiler optimizations, known as *data-flow optimizations*, has been developed to analyze the flow of data through the program and removes redundancies across these constructs. They are effective in generating code that resembles code written by a skilled programmer at a lower level.

Object orientation was first introduced in Simula in 1967, and has been incorporated in languages such as Smalltalk, C++, C#, and Java. The key ideas behind object orientation are

1. Data abstraction and
2. Inheritance of properties,

both of which have been found to make programs more modular and easier to maintain. Object-oriented programs are different from those written in many other languages, in that they consist of many more, but smaller, procedures (called *methods* in object-oriented terms). Thus, compiler optimizations must be able to perform well across the procedural boundaries of the source program. Procedure inlining, which is the replacement of a procedure call by the body of the procedure, is particularly useful here. Optimizations to speed up virtual method dispatches have also been developed.

Java has many features that make programming easier, many of which have been introduced previously in other languages. The Java language is type-safe; that is, an object cannot be used as an object of an unrelated type. All array accesses are checked to ensure that they lie within the bounds of the array. Java has no pointers and does not allow pointer arithmetic. It has a built-in garbage-collection facility that automatically frees the memory of variables that are no longer in use. While all these features make programming easier, they incur a run-time overhead. Compiler optimizations have been developed to reduce the overhead, for example, by eliminating unnecessary range checks and by allocating objects that are not accessible beyond a procedure on the stack instead of the heap. Effective algorithms also have been developed to minimize the overhead of garbage collection.

In addition, Java is designed to support portable and mobile code. Programs are distributed as Java bytecode, which must either be interpreted or compiled into native code dynamically, that is, at run time. Dynamic compilation has also been studied in other contexts, where information is extracted dynamically at run time and used to produce better-optimized code. In dynamic optimization, it is important to minimize the compilation time as it is part of the execution overhead. A common technique used is to only compile and optimize those parts of the program that will be frequently executed.

### 1.5.2 Optimizations for Computer Architectures

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies*. Parallelism can be found at several levels: at the *instruction level*, where multiple operations are executed simultaneously and at the *processor level*, where different threads of the same application are run on different processors. Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

#### Parallelism

All modern microprocessors exploit instruction-level parallelism. However, this parallelism can be hidden from the programmer. Programs are written as if all instructions were executed in sequence; the hardware dynamically checks for dependencies in the sequential instruction stream and issues them in parallel when possible. In some cases, the machine includes a hardware scheduler that can change the instruction ordering to increase the parallelism in the program. Whether the hardware reorders the instructions or not, compilers can rearrange the instructions to make instruction-level parallelism more effective.

Instruction-level parallelism can also appear explicitly in the instruction set. VLIW (Very Long Instruction Word) machines have instructions that can issue

multiple operations in parallel. The Intel IA64 is a well-known example of such an architecture. All high-performance, general-purpose microprocessors also include instructions that can operate on a vector of data at the same time. Compiler techniques have been developed to generate code automatically for such machines from sequential programs.

Multiprocessors have also become prevalent; even personal computers often have multiple processors. Programmers can write multithreaded code for multiprocessors, or parallel code can be automatically generated by a compiler from conventional sequential programs. Such a compiler hides from the programmers the details of finding parallelism in a program, distributing the computation across the machine, and minimizing synchronization and communication among the processors. Many scientific-computing and engineering applications are computation-intensive and can benefit greatly from parallel processing. Parallelization techniques have been developed to translate automatically sequential scientific programs into multiprocessor code.

### Memory Hierarchies

A memory hierarchy consists of several levels of storage with different speeds and sizes, with the level closest to the processor being the fastest but smallest. The average memory-access time of a program is reduced if most of its accesses are satisfied by the faster levels of the hierarchy. Both parallelism and the existence of a memory hierarchy improve the potential performance of a machine, but they must be harnessed effectively by the compiler to deliver real performance on an application.

Memory hierarchies are found in all machines. A processor usually has a small number of registers consisting of hundreds of bytes, several levels of caches containing kilobytes to megabytes, physical memory containing megabytes to gigabytes, and finally secondary storage that contains gigabytes and beyond. Correspondingly, the speed of accesses between adjacent levels of the hierarchy can differ by two or three orders of magnitude. The performance of a system is often limited not by the speed of the processor but by the performance of the memory subsystem. While compilers traditionally focus on optimizing the processor execution, more emphasis is now placed on making the memory hierarchy more effective.

Using registers effectively is probably the single most important problem in optimizing a program. Unlike registers that have to be managed explicitly in software, caches and physical memories are hidden from the instruction set and are managed by hardware. It has been found that cache-management policies implemented by hardware are not effective in some cases, especially in scientific code that has large data structures (arrays, typically). It is possible to improve the effectiveness of the memory hierarchy by changing the layout of the data, or changing the order of instructions accessing the data. We can also change the layout of code to improve the effectiveness of instruction caches.

### 1.5.3 Design of New Computer Architectures

In the early days of computer architecture design, compilers were developed after the machines were built. That has changed. Since programming in high-level languages is the norm, the performance of a computer system is determined not by its raw speed but also by how well compilers can exploit its features. Thus, in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features.

#### RISC

One of the best known examples of how compilers influenced the design of computer architecture was the invention of the RISC (Reduced Instruction-Set Computer) architecture. Prior to this invention, the trend was to develop progressively complex instruction sets intended to make assembly programming easier; these architectures were known as CISC (Complex Instruction-Set Computer). For example, CISC instruction sets include complex memory-addressing modes to support data-structure accesses and procedure-invocation instructions that save registers and pass parameters on the stack.

Compiler optimizations often can reduce these instructions to a small number of simpler operations by eliminating the redundancies across complex instructions. Thus, it is desirable to build simple instruction sets; compilers can use them effectively and the hardware is much easier to optimize.

Most general-purpose processor architectures, including PowerPC, SPARC, MIPS, Alpha, and PA-RISC, are based on the RISC concept. Although the x86 architecture—the most popular microprocessor—has a CISC instruction set, many of the ideas developed for RISC machines are used in the implementation of the processor itself. Moreover, the most effective way to use a high-performance x86 machine is to use just its simple instructions.

#### Specialized Architectures

Over the last three decades, many architectural concepts have been proposed. They include data flow machines, vector machines, VLIW (Very Long Instruction Word) machines, SIMD (Single Instruction, Multiple Data) arrays of processors, systolic arrays, multiprocessors with shared memory, and multiprocessors with distributed memory. The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.

Some of these ideas have made their way into the designs of embedded machines. Since entire systems can fit on a single chip, processors need no longer be prepackaged commodity units, but can be tailored to achieve better cost-effectiveness for a particular application. Thus, in contrast to general-purpose processors, where economies of scale have led computer architectures

to converge, application-specific processors exhibit a diversity of computer architectures. Compiler technology is needed not only to support programming for these architectures, but also to evaluate proposed architectural designs.

### 1.5.4 Program Translations

While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages. The following are some of the important applications of program-translation techniques.

#### Binary Translation

Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originally compiled for another instruction set. Binary translation technology has been used by various computer companies to increase the availability of software for their machines. In particular, because of the domination of the x86 personal-computer market, most software titles are available as x86 code. Binary translators have been developed to convert x86 code into both Alpha and Sparc code. Binary translation was also used by Transmeta Inc. in their implementation of the x86 instruction set. Instead of executing the complex x86 instruction set directly in hardware, the Transmeta Crusoe processor is a VLIW processor that relies on binary translation to convert x86 code into native VLIW code.

Binary translation can also be used to provide backward compatibility. When the processor in the Apple Macintosh was changed from the Motorola MC 68040 to the PowerPC in 1994, binary translation was used to allow PowerPC processors run legacy MC 68040 code.

#### Hardware Synthesis

Not only is most software written in high-level languages; even hardware designs are mostly described in high-level hardware description languages like Verilog and VHDL (Very high-speed integrated circuit Hardware Description Language). Hardware designs are typically described at the register transfer level (RTL), where variables represent registers and expressions represent combinational logic. Hardware-synthesis tools translate RTL descriptions automatically into gates, which are then mapped to transistors and eventually to a physical layout. Unlike compilers for programming languages, these tools often take hours optimizing the circuit. Techniques to translate designs at higher levels, such as the behavior or functional level, also exist.

#### Database Query Interpreters

Besides specifying software and hardware, languages are useful in many other applications. For example, query languages, especially SQL (Structured Query

Language), are used to search databases. Database queries consist of predicates containing relational and boolean operators. They can be interpreted or compiled into commands to search a database for records satisfying that predicate.

#### Compiled Simulation

Simulation is a general technique used in many scientific and engineering disciplines to understand a phenomenon or to validate a design. Inputs to a simulator usually include the description of the design and specific input parameters for that particular simulation run. Simulations can be very expensive. We typically need to simulate many possible design alternatives on many different input sets, and each experiment may take days to complete on a high-performance machine. Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code that simulates that particular design natively. Compiled simulation can run orders of magnitude faster than an interpreter-based approach. Compiled simulation is used in many state-of-the-art tools that simulate designs written in Verilog or VHDL.

### 1.5.5 Software Productivity Tools

Programs are arguably the most complicated engineering artifacts ever produced; they consist of many many details, every one of which must be correct before the program will work completely. As a result, errors are rampant in programs; errors may crash a system, produce wrong results, render a system vulnerable to security attacks, or even lead to catastrophic failures in critical systems. Testing is the primary technique for locating errors in programs.

An interesting and promising complementary approach is to use data-flow analysis to locate errors statically (that is, before the program is run). Data-flow analysis can find errors along all the possible execution paths, and not just those exercised by the input data sets, as in the case of program testing. Many of the data-flow-analysis techniques, originally developed for compiler optimizations, can be used to create tools that assist programmers in their software engineering tasks.

The problem of finding all program errors is undecidable. A data-flow analysis may be designed to warn the programmers of all possible statements with a particular category of errors. But if most of these warnings are false alarms, users will not use the tool. Thus, practical error detectors are often neither sound nor complete. That is, they may not find all the errors in the program, and not all errors reported are guaranteed to be real errors. Nonetheless, various static analyses have been developed and shown to be effective in finding errors, such as dereferencing null or freed pointers, in real programs. The fact that error detectors may be unsound makes them significantly different from compiler optimizations. Optimizers must be conservative and cannot alter the semantics of the program under any circumstances.

In the balance of this section, we shall mention several ways in which program analysis, building upon techniques originally developed to optimize code in compilers, have improved software productivity. Of special importance are techniques that detect statically when a program might have a security vulnerability.

### Type Checking

Type checking is an effective and well-established technique to catch inconsistencies in programs. It can be used to catch errors, for example, where an operation is applied to the wrong type of object, or if parameters passed to a procedure do not match the signature of the procedure. Program analysis can go beyond finding type errors by analyzing the flow of data through a program. For example, if a pointer is assigned `null` and then immediately dereferenced, the program is clearly in error.

The same technology can be used to catch a variety of security holes, in which an attacker supplies a string or other data that is used carelessly by the program. A user-supplied string can be labeled with a type "dangerous." If this string is not checked for proper format, then it remains "dangerous," and if a string of this type is able to influence the control-flow of the code at some point in the program, then there is a potential security flaw.

### Bounds Checking

It is easier to make mistakes when programming in a lower-level language than a higher-level one. For example, many security breaches in systems are caused by buffer overflows in programs written in C. Because C does not have array-bounds checks, it is up to the user to ensure that the arrays are not accessed out of bounds. Failing to check that the data supplied by the user can overflow a buffer, the program may be tricked into storing user data outside of the buffer. An attacker can manipulate the input data that causes the program to misbehave and compromise the security of the system. Techniques have been developed to find buffer overflows in programs, but with limited success.

Had the program been written in a safe language that includes automatic range checking, this problem would not have occurred. The same data-flow analysis that is used to eliminate redundant range checks can also be used to locate buffer overflows. The major difference, however, is that failing to eliminate a range check would only result in a small run-time cost, while failing to identify a potential buffer overflow may compromise the security of the system. Thus, while it is adequate to use simple techniques to optimize range checks, sophisticated analyses, such as tracking the values of pointers across procedures, are needed to get high-quality results in error detection tools.

### Memory-Management Tools

Garbage collection is another excellent example of the tradeoff between efficiency and a combination of ease of programming and software reliability. Automatic memory management obliterates all memory-management errors (e.g., "memory leaks"), which are a major source of problems in C and C++ programs. Various tools have been developed to help programmers find memory management errors. For example, Purify is a widely used tool that dynamically catches memory management errors as they occur. Tools that help identify some of these problems statically have also been developed.

## 1.6 Programming Language Basics

In this section, we shall cover the most important terminology and distinctions that appear in the study of programming languages. It is not our purpose to cover all concepts or all the popular programming languages. We assume that the reader is familiar with at least one of C, C++, C#, or Java, and may have encountered other languages as well.

### 1.6.1 The Static/Dynamic Distinction

Among the most important issues that we face when designing a compiler for a language is what decisions can the compiler make about a program. If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a *static* policy or that the issue can be decided at *compile time*. On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a *dynamic policy* or to require a decision at *run time*.

One issue on which we shall concentrate is the scope of declarations. The *scope* of a declaration of $x$ is the region of the program in which uses of $x$ refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope*. With dynamic scope, as the program runs, the same use of $x$ could refer to any of several different declarations of $x$.

Most languages, such as C and Java, use static scope. We shall discuss static scoping in Section 1.6.3.

**Example 1.3:** As another example of the static/dynamic distinction, consider the use of the term "static" as it applies to data in a Java class declaration. In Java, a variable is a name for a location in memory used to hold a data value. Here, "static" refers not to the scope of the variable, but rather to the ability of the compiler to determine the location in memory where the declared variable can be found. A declaration like

```
public static int x;
```

makes $x$ a *class variable* and says that there is only one copy of $x$, no matter how many objects of this class are created. Moreover, the compiler can determine a location in memory where this integer $x$ will be held. In contrast, had "static" been omitted from this declaration, then each object of the class would have its own location where $x$ would be held, and the compiler could not determine all these places in advance of running the program. □

### 1.6.2 Environments and States

Another important distinction we must make when discussing programming languages is whether changes occurring as the program runs affect the values of data elements or affect the interpretation of names for that data. For example, the execution of an assignment such as `x = y + 1` changes the value denoted by the name $x$. More specifically, the assignment changes the value in whatever location is denoted by $x$.

It may be less clear that the location denoted by $x$ can change at run time. For instance, as we discussed in Example 1.3, if $x$ is not a static (or "class") variable, then every object of the class has its own location for an instance of variable $x$. In that case, the assignment to $x$ can change any of those "instance" variables, depending on the object to which a method containing that assignment is applied.

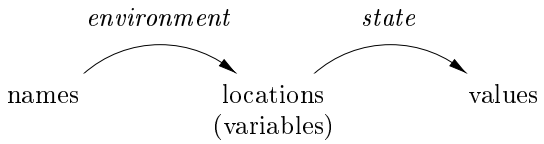



Figure 1.8: Two-stage mapping from names to values

The association of names with locations in memory (the *store*) and then with values can be described by two mappings that change as the program runs (see Fig. 1.8):

1. The *environment* is a mapping from names to locations in the store. Since variables refer to locations ("l-values" in the terminology of C), we could alternatively define an environment as a mapping from names to variables.

2. The *state* is a mapping from locations in store to their values. That is, the state maps l-values to their corresponding r-values, in the terminology of C.

Environments change according to the scope rules of a language.

**Example 1.4:** Consider the C program fragment in Fig. 1.9. Integer $i$ is declared a global variable, and also declared as a variable local to function $f$. When $f$ is executing, the environment adjusts so that name $i$ refers to the

```
...
int i;                  /* global i       */
...
void f(···) {
      int i;            /* local i        */
      ...
      i = 3;            /* use of local i */
      ...
}
...
      x = i + 1;        /* use of global i */
```

Figure 1.9: Two declarations of the name $i$

location reserved for the $i$ that is local to $f$, and any use of $i$, such as the assignment `i = 3` shown explicitly, refers to that location. Typically, the local $i$ is given a place on the run-time stack.

Whenever a function $g$ other than $f$ is executing, uses of $i$ cannot refer to the $i$ that is local to $f$. Uses of name $i$ in $g$ must be within the scope of some other declaration of $i$. An example is the explicitly shown statement `x = i+1`, which is inside some procedure whose definition is not shown. The $i$ in $i + 1$ presumably refers to the global $i$. As in most languages, declarations in C must precede their use, so a function that comes before the global $i$ cannot refer to it. □

The environment and state mappings in Fig. 1.8 are dynamic, but there are a few exceptions:

1. *Static versus dynamic binding* of names to locations. Most binding of names to locations is dynamic, and we discuss several approaches to this binding throughout the section. Some declarations, such as the global $i$ in Fig. 1.9, can be given a location in the store once and for all, as the compiler generates object code.[2]

2. *Static versus dynamic binding* of locations to values. The binding of locations to values (the second stage in Fig. 1.8), is generally dynamic as well, since we cannot tell the value in a location until we run the program. Declared constants are an exception. For instance, the C definition

```
#define ARRAYSIZE 1000
```

[2]Technically, the C compiler will assign a location in virtual memory for the global $i$, leaving it to the loader and the operating system to determine where in the physical memory of the machine $i$ will be located. However, we shall not worry about "relocation" issues such as these, which have no impact on compiling. Instead, we treat the address space that the compiler uses for its output code as if it gave physical memory locations.

### Names, Identifiers, and Variables

Although the terms "name" and "variable," often refer to the same thing, we use them carefully to distinguish between compile-time names and the run-time locations denoted by names.

An *identifier* is a string of characters, typically letters or digits, that refers to (identifies) an entity, such as a data object, a procedure, a class, or a type. All identifiers are names, but not all names are identifiers. Names can also be expressions. For example, the name $x.y$ might denote the field $y$ of a structure denoted by $x$. Here, $x$ and $y$ are identifiers, while $x.y$ is a name, but not an identifier. Composite names like $x.y$ are called *qualified* names.

A *variable* refers to a particular location of the store. It is common for the same identifier to be declared more than once; each such declaration introduces a new variable. Even if each identifier is declared just once, an identifier local to a recursive procedure will refer to different locations of the store at different times.

binds the name `ARRAYSIZE` to the value 1000 statically. We can determine this binding by looking at the statement, and we know that it is impossible for this binding to change when the program executes.

### 1.6.3 Static Scope and Block Structure

Most languages, including C and its family, use static scope. The scope rules for C are based on program structure; the scope of a declaration is determined implicitly by where the declaration appears in the program. Later languages, such as C++, Java, and C#, also provide explicit control over scopes through the use of keywords like **public**, **private**, and **protected**.

In this section we consider static-scope rules for a language with blocks, where a *block* is a grouping of declarations and statements. C uses braces { and } to delimit a block; the alternative use of **begin** and **end** for the same purpose dates back to Algol.

**Example 1.5:** To a first approximation, the C static-scope policy is as follows:

1. A C program consists of a sequence of top-level declarations of variables and functions.

2. Functions may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears.

### Procedures, Functions, and Methods

To avoid saying "procedures, functions, or methods," each time we want to talk about a subprogram that may be called, we shall usually refer to all of them as "procedures." The exception is that when talking explicitly of programs in languages like C that have only functions, we shall refer to them as "functions." Or, if we are discussing a language like Java that has only methods, we shall use that term instead.

A function generally returns a value of some type (the "return type"), while a procedure does not return any value. C and similar languages, which have only functions, treat procedures as functions that have a special return type "void," to signify no return value. Object-oriented languages like Java and C++ use the term "methods." These can behave like either functions or procedures, but are associated with a particular class.

3. The scope of a top-level declaration of a name $x$ consists of the entire program that follows, with the exception of those statements that lie within a function that also has a declaration of $x$.

The additional detail regarding the C static-scope policy deals with variable declarations within statements. We examine such declarations next and in Example 1.6. □

In C, the syntax of blocks is given by

1. One type of statement is a block. Blocks can appear anywhere that other types of statements, such as assignment statements, can appear.

2. A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces.

Note that this syntax allows blocks to be nested inside each other. This nesting property is referred to as *block structure*. The C family of languages has block structure, except that a function may not be defined inside another function.

We say that a declaration $D$ "belongs" to a block $B$ if $B$ is the most closely nested block containing $D$; that is, $D$ is located within $B$, but not within any block that is nested within $B$.

The static-scope rule for variable declarations in block-structured languages is as follows. If declaration $D$ of name $x$ belongs to block $B$, then the scope of $D$ is all of $B$, except for any blocks $B'$ nested to any depth within $B$, in which $x$ is redeclared. Here, $x$ is redeclared in $B'$ if some other declaration $D'$ of the same name $x$ belongs to $B'$.

An equivalent way to express this rule is to focus on a use of a name $x$. Let $B_1, B_2, \ldots, B_k$ be all the blocks that surround this use of $x$, with $B_k$ the smallest, nested within $B_{k-1}$, which is nested within $B_{k-2}$, and so on. Search for the largest $i$ such that there is a declaration of $x$ belonging to $B_i$. This use of $x$ refers to the declaration in $B_i$. Alternatively, this use of $x$ is within the scope of the declaration in $B_i$.

```
main() {
    int a = 1;                                          B1
    int b = 1;
    {
        int b = 2;                                      B2
        {
            int a = 3;                                  B3
            cout << a << b;
        }
        {
            int b = 4;                                  B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Figure 1.10: Blocks in a C++ program

**Example 1.6:** The C++ program in Fig. 1.10 has four blocks, with several definitions of variables $a$ and $b$. As a memory aid, each declaration initializes its variable to the number of the block to which it belongs.

For instance, consider the declaration `int a = 1` in block $B_1$. Its scope is all of $B_1$, except for those blocks nested (perhaps deeply) within $B_1$ that have their own declaration of $a$. $B_2$, nested immediately within $B_1$, does not have a declaration of $a$, but $B_3$ does. $B_4$ does not have a declaration of $a$, so block $B_3$ is the only place in the entire program that is outside the scope of the declaration of the name $a$ that belongs to $B_1$. That is, this scope includes $B_4$ and all of $B_2$ except for the part of $B_2$ that is within $B_3$. The scopes of all five declarations are summarized in Fig. 1.11.

From another point of view, let us consider the output statement in block $B_4$ and bind the variables $a$ and $b$ used there to the proper declarations. The list of surrounding blocks, in order of increasing size, is $B_4, B_2, B_1$. Note that $B_3$ does not surround the point in question. $B_4$ has a declaration of $b$, so it is to this declaration that this use of $b$ refers, and the value of $b$ printed is 4. However, $B_4$ does not have a declaration of $a$, so we next look at $B_2$. That block does not have a declaration of $a$ either, so we proceed to $B_1$. Fortunately,

| DECLARATION | SCOPE |
|---|---|
| `int a = 1;` | $B_1 - B_3$ |
| `int b = 1;` | $B_1 - B_2$ |
| `int b = 2;` | $B_2 - B_4$ |
| `int a = 3;` | $B_3$ |
| `int b = 4;` | $B_4$ |

Figure 1.11: Scopes of declarations in Example 1.6

there is a declaration `int a = 1` belonging to that block, so the value of $a$ printed is 1. Had there been no such declaration, the program would have been erroneous. □

### 1.6.4 Explicit Access Control

Classes and structures introduce a new scope for their members. If $p$ is an object of a class with a field (member) $x$, then the use of $x$ in $p.x$ refers to field $x$ in the class definition. In analogy with block structure, the scope of a member declaration $x$ in a class $C$ extends to any subclass $C'$, except if $C'$ has a local declaration of the same name $x$.

Through the use of keywords like **public**, **private**, and **protected**, object-oriented languages such as C++ or Java provide explicit control over access to member names in a superclass. These keywords support *encapsulation* by restricting access. Thus, private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C++ term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

In C++, a class definition may be separated from the definitions of some or all of its methods. Therefore, a name $x$ associated with the class $C$ may have a region of the code that is outside its scope, followed by another region (a method definition) that is within its scope. In fact, regions inside and outside the scope may alternate, until all the methods have been defined.

### 1.6.5 Dynamic Scope

Technically, any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope*, however, usually refers to the following policy: a use of a name $x$ refers to the declaration of $x$ in the most recently called, not-yet-terminated, procedure with such a declaration. Dynamic scoping of this type appears only in special situations. We shall consider two examples of dynamic policies: macro expansion in the C preprocessor and method resolution in object-oriented programming.

### Declarations and Definitions

The apparently similar terms "declaration" and "definition" for programming-language concepts are actually quite different. Declarations tell us about the types of things, while definitions tell us about their values. Thus, `int i` is a declaration of $i$, while `i = 1` is a definition of $i$.

The difference is more significant when we deal with methods or other procedures. In C++, a method is declared in a class definition, by giving the types of the arguments and result of the method (often called the *signature* for the method). The method is then defined, i.e., the code for executing the method is given, in another place. Similarly, it is common to define a C function in one file and declare it in other files where the function is used.

**Example 1.7:** In the C program of Fig. 1.12, identifier $a$ is a macro that stands for expression $(x + 1)$. But what is $x$? We cannot resolve $x$ statically, that is, in terms of the program text.

```
#define a (x+1)

int x = 2;

void b() { int x = 1; printf("%d\n", a); }

void c() { printf("%d\n", a); }

void main() { b(); c(); }
```

Figure 1.12: A macro whose names must be scoped dynamically

In fact, in order to interpret $x$, we must use the usual dynamic-scope rule. We examine all the function calls that are currently active, and we take the most recently called function that has a declaration of $x$. It is to this declaration that the use of $x$ refers.

In the example of Fig. 1.12, the function *main* first calls function $b$. As $b$ executes, it prints the value of the macro $a$. Since $(x + 1)$ must be substituted for $a$, we resolve this use of $x$ to the declaration `int x=1` in function $b$. The reason is that $b$ has a declaration of $x$, so the $(x + 1)$ in the `printf` in $b$ refers to this $x$. Thus, the value printed is 2.

After $b$ finishes, and $c$ is called, we again need to print the value of macro $a$. However, the only $x$ accessible to $c$ is the global $x$. The `printf` statement in $c$ thus refers to this declaration of $x$, and value 3 is printed. □

Dynamic scope resolution is also essential for polymorphic procedures, those that have two or more definitions for the same name, depending only on the

### Analogy Between Static and Dynamic Scoping

While there could be any number of static or dynamic policies for scoping, there is an interesting relationship between the normal (block-structured) static scoping rule and the normal dynamic policy. In a sense, the dynamic rule is to time as the static rule is to space. While the static rule asks us to find the declaration whose unit (block) most closely surrounds the physical location of the use, the dynamic rule asks us to find the declaration whose unit (procedure invocation) most closely surrounds the time of the use.

types of the arguments. In some languages, such as ML (see Section 7.3.3), it is possible to determine statically types for all uses of names, in which case the compiler can replace each use of a procedure name $p$ by a reference to the code for the proper procedure. However, in other languages, such as Java and C++, there are times when the compiler cannot make that determination.

**Example 1.8:** A distinguishing feature of object-oriented programming is the ability of each object to invoke the appropriate method in response to a message. In other words, the procedure called when $x.m()$ is executed depends on the class of the object denoted by $x$ at that time. A typical example is as follows:

1. There is a class $C$ with a method named $m()$.
2. $D$ is a subclass of $C$, and $D$ has its own method named $m()$.
3. There is a use of $m$ of the form $x.m()$, where $x$ is an object of class $C$.

Normally, it is impossible to tell at compile time whether $x$ will be of class $C$ or of the subclass $D$. If the method application occurs several times, it is highly likely that some will be on objects denoted by $x$ that are in class $C$ but not $D$, while others will be in class $D$. It is not until run-time that it can be decided which definition of $m$ is the right one. Thus, the code generated by the compiler must determine the class of the object $x$, and call one or the other method named $m$. □

### 1.6.6 Parameter Passing Mechanisms

All programming languages have a notion of a procedure, but they can differ in how these procedures get their arguments. In this section, we shall consider how the *actual parameters* (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by-reference," or both. We shall explain these terms, and another method known as "call-by-name," that is primarily of historical interest.

### Call-by-Value

In *call-by-value*, the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. This method is used in C and Java, and is a common option in C++, as well as in most other languages. Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

Note, however, that in C we can pass a pointer to a variable to allow that variable to be changed by the callee. Likewise, array names passed as parameters in C, C++, or Java give the called procedure what is in effect a pointer or reference to the array itself. Thus, if $a$ is the name of an array of the calling procedure, and it is passed by value to corresponding formal parameter $x$, then an assignment such as `x[i] = 2` really changes the array element $a[i]$ to 2. The reason is that, although $x$ gets a copy of the value of $a$, that value is really a pointer to the beginning of the area of the store where the array named $a$ is located.

Similarly, in Java, many variables are really references, or pointers, to the things they stand for. This observation applies to arrays, strings, and objects of all classes. Even though Java uses call-by-value exclusively, whenever we pass the name of an object to a called procedure, the value received by that procedure is in effect a pointer to the object. Thus, the called procedure is able to affect the value of the object itself.

### Call-by-Reference

In *call-by-reference*, the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change the value in this location, but can have no effect on the data of the caller.

Call-by-reference is used for "ref" parameters in C++ and is an option in many other languages. It is almost essential when the formal parameter is a large object, array, or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large. As we noted when discussing call-by-value, languages such as Java solve the problem of passing arrays, strings, or other objects by copying only a reference to those objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.

### Call-by-Name

A third mechanism — call-by-name — was used in the early programming language Algol 60. It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct). When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

## 1.6.7 Aliasing

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well.

**Example 1.9:** Suppose $a$ is an array belonging to a procedure $p$, and $p$ calls another procedure $q(x, y)$ with a call $q(a, a)$. Suppose also that parameters are passed by value, but that array names are really references to the location where the array is stored, as in C or similar languages. Now, $x$ and $y$ have become aliases of each other. The important point is that if within $q$ there is an assignment `x[10] = 2`, then the value of $y[10]$ also becomes 2. □

It turns out that understanding aliasing and the mechanisms that create it is essential if a compiler is to optimize a program. As we shall see starting in Chapter 9, there are many situations where we can only optimize code if we can be sure certain variables are not aliased. For instance, we might determine that `x = 2` is the only place that variable $x$ is ever assigned. If so, then we can replace a use of $x$ by a use of 2; for example, replace `a = x+3` by the simpler `a = 5`. But suppose there were another variable $y$ that was aliased to $x$. Then an assignment `y = 4` might have the unexpected effect of changing $x$. It might also mean that replacing `a = x+3` by `a = 5` was a mistake; the proper value of $a$ could be 7 there.

## 1.6.8 Exercises for Section 1.6

**Exercise 1.6.1:** For the block-structured C code of Fig. 1.13(a), indicate the values assigned to $w$, $x$, $y$, and $z$.

**Exercise 1.6.2:** Repeat Exercise 1.6.1 for the code of Fig. 1.13(b).

**Exercise 1.6.3:** For the block-structured code of Fig. 1.14, assuming the usual static scoping of declarations, give the scope for each of the twelve declarations.

```
int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;
```

(a) Code for Exercise 1.6.1

```
int w, x, y, z;
int i = 3; int j = 4;
{   int i = 5;
    w = i + j;
}
x = i + j;
{   int j = 6;
    i = 7;
    y = i + j;
}
z = i + j;
```

(b) Code for Exercise 1.6.2

Figure 1.13: Block-structured code

```
{   int w, x, y, z;     /* Block B1 */
    {   int x, z;       /* Block B2 */
        {   int w, x;   /* Block B3 */ }
    }
    {   int w, x;       /* Block B4 */
        {   int y, z;   /* Block B5 */ }
    }
}
```

Figure 1.14: Block structured code for Exercise 1.6.3

**Exercise 1.6.4:** What is printed by the following C code?

```
#define a (x+1)
int x = 2;
void b() { x = a; printf("%d\n", x); }
void c() { int x = 1; printf("%d\n", a); }
void main() { b(); c(); }
```

## 1.7 Summary of Chapter 1

- *Language Processors.* An integrated software development environment includes many different kinds of language processors such as compilers, interpreters, assemblers, linkers, loaders, debuggers, profilers.

- *Compiler Phases.* A compiler operates as a sequence of phases, each of which transforms the source program from one intermediate representation to another.

## 1.7. SUMMARY OF CHAPTER 1

- *Machine and Assembly Languages.* Machine languages were the first-generation programming languages, followed by assembly languages. Programming in these languages was time consuming and error prone.
- *Modeling in Compiler Design.* Compiler design is one of the places where theory has had the most impact on practice. Models that have been found useful include automata, grammars, regular expressions, trees, and many others.
- *Code Optimization.* Although code cannot truly be "optimized," the science of improving the efficiency of code is both complex and very important. It is a major portion of the study of compilation.
- *Higher-Level Languages.* As time goes on, programming languages take on progressively more of the tasks that formerly were left to the programmer, such as memory management, type-consistency checking, or parallel execution of code.
- *Compilers and Computer Architecture.* Compiler technology influences computer architecture, as well as being influenced by the advances in architecture. Many modern innovations in architecture depend on compilers being able to extract from source programs the opportunities to use the hardware capabilities effectively.
- *Software Productivity and Software Security.* The same technology that allows compilers to optimize code can be used for a variety of program-analysis tasks, ranging from detecting common program bugs to discovering that a program is vulnerable to one of the many kinds of intrusions that "hackers" have discovered.
- *Scope Rules.* The *scope* of a declaration of $x$ is the context in which uses of $x$ refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope*.
- *Environments.* The association of names with locations in memory and then with values can be described in terms of *environments*, which map names to locations in store, and *states*, which map locations to their values.
- *Block Structure.* Languages that allow blocks to be nested are said to have *block structure*. A name $x$ in a nested block $B$ is in the scope of a declaration $D$ of $x$ in an enclosing block if there is no other declaration of $x$ in an intervening block.
- *Parameter Passing.* Parameters are passed from a calling procedure to the callee either by value or by reference. When large objects are passed by value, the values passed are really references to the objects themselves, resulting in an effective call-by-reference.

- *Aliasing.* When parameters are (effectively) passed by reference, two formal parameters can refer to the same object. This possibility allows a change in one variable to change another.

## 1.8 References for Chapter 1

For the development of programming languages that were created and in use by 1967, including Fortran, Algol, Lisp, and Simula, see [7]. For languages that were created by 1982, including C, C++, Pascal, and Smalltalk, see [1].

The GNU Compiler Collection, gcc, is a popular source of open-source compilers for C, C++, Fortran, Java, and other languages [2]. Phoenix is a compiler-construction toolkit that provides an integrated framework for building the program analysis, code generation, and code optimization phases of compilers discussed in this book [3].

For more information about programming language concepts, we recommend [5,6]. For more on computer architecture and how it impacts compiling, we suggest [4].

1. Bergin, T. J. and R. G. Gibson, *History of Programming Languages*, ACM Press, New York, 1996.

2. `http://gcc.gnu.org/` .

3. `http://research.microsoft.com/phoenix/default.aspx` .

4. Hennessy, J. L. and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan-Kaufmann, San Francisco, CA, 2004.

5. Scott, M. L., *Programming Language Pragmatics, second edition*, Morgan-Kaufmann, San Francisco, CA, 2006.

6. Sethi, R., *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1996.

7. Wexelblat, R. L., *History of Programming Languages*, Academic Press, New York, 1981.

# Chapter 2

# A Simple Syntax-Directed Translator

This chapter is an introduction to the compiling techniques in Chapters 3 through 6 of this book. It illustrates the techniques by developing a working Java program that translates representative programming language statements into three-address code, an intermediate representation. In this chapter, the emphasis is on the front end of a compiler, in particular on lexical analysis, parsing, and intermediate code generation. Chapters 7 and 8 show how to generate machine instructions from three-address code.

We start small by creating a syntax-directed translator that maps infix arithmetic expressions into postfix expressions. We then extend this translator to map code fragments as shown in Fig. 2.1 into three-address code of the form in Fig. 2.2.

The working Java translator appears in Appendix A. The use of Java is convenient, but not essential. In fact, the ideas in this chapter predate the creation of both Java and C.

```
{
      int i; int j; float[100] a; float v; float x;

      while ( true ) {
            do i = i+1; while ( a[i] < v );
            do j = j-1; while ( a[j] > v );
            if ( i >= j ) break;
            x = a[i]; a[i] = a[j]; a[j] = x;
      }
}
```

Figure 2.1: A code fragment to be translated

```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

Figure 2.2: Simplified intermediate code for the program fragment in Fig. 2.1

## 2.1 Introduction

The analysis phase of a compiler breaks up a source program into constituent pieces and produces an internal representation for it, called intermediate code. The synthesis phase translates the intermediate code into the target program.

Analysis is organized around the "syntax" of the language to be compiled. The *syntax* of a programming language describes the proper form of its programs, while the *semantics* of the language defines what its programs mean; that is, what each program does when it executes. For specifying syntax, we present a widely used notation, called context-free grammars or BNF (for Backus-Naur Form) in Section 2.2. With the notations currently available, the semantics of a language is much more difficult to describe than the syntax. For specifying semantics, we shall therefore use informal descriptions and suggestive examples.

Besides specifying the syntax of a language, a context-free grammar can be used to help guide the translation of programs. In Section 2.3, we introduce a grammar-oriented compiling technique known as *syntax-directed translation*. Parsing or syntax analysis is introduced in Section 2.4.

The rest of this chapter is a quick tour through the model of a compiler front end in Fig. 2.3. We begin with the parser. For simplicity, we consider the syntax-directed translation of infix expressions to postfix form, a notation in which operators appear after their operands. For example, the postfix form of the expression $9-5+2$ is $95-2+$. Translation into postfix form is rich enough to illustrate syntax analysis, yet simple enough that the translator is shown in full in Section 2.5. The simple translator handles expressions like $9-5+2$, consisting of digits separated by plus and minus signs. One reason for starting with such simple expressions is that the syntax analyzer can work directly with the individual characters for operators and operands.

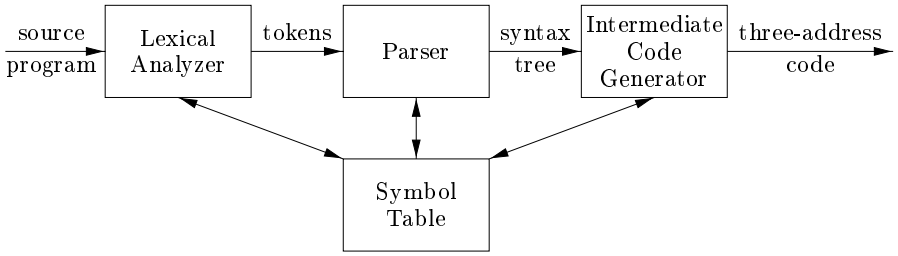
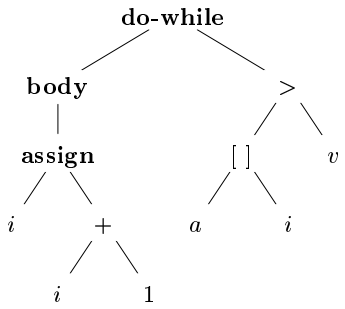



Figure 2.3: A model of a compiler front end

A lexical analyzer allows a translator to handle multicharacter constructs like identifiers, which are written as sequences of characters, but are treated as units called *tokens* during syntax analysis; for example, in the expression `count + 1`, the identifier `count` is treated as a unit. The lexical analyzer in Section 2.6 allows numbers, identifiers, and "white space" (blanks, tabs, and newlines) to appear within expressions.

Next, we consider intermediate-code generation. Two forms of intermediate code are illustrated in Fig. 2.4. One form, called *abstract syntax trees* or simply *syntax trees*, represents the hierarchical syntactic structure of the source program. In the model in Fig. 2.3, the parser produces a syntax tree, that is further translated into three-address code. Some compilers combine parsing and intermediate-code generation into one component.




(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

Figure 2.4: Intermediate code for "`do i = i + 1; while (a[i] < v);`"

The root of the abstract syntax tree in Fig. 2.4(a) represents an entire do-while loop. The left child of the root represents the body of the loop, which consists of only the assignment `i = i + 1;`. The right child of the root represents the condition `a[i] < v`. An implementation of syntax trees appears in Section 2.8.

The other common intermediate representation, shown in Fig. 2.4(b), is a

sequence of "three-address" instructions; a more complete example appears in Fig. 2.2. This form of intermediate code takes its name from instructions of the form $x = y$ **op** $z$, where **op** is a binary operator, $y$ and $z$ are the addresses for the operands, and $x$ is the address for the result of the operation. A three-address instruction carries out at most one operation, typically a computation, a comparison, or a branch.

In Appendix A, we put the techniques in this chapter together to build a compiler front end in Java. The front end translates statements into assembly-level instructions.

## 2.2 Syntax Definition

In this section, we introduce a notation — the "context-free grammar," or "grammar" for short — that is used to specify the syntax of a language. Grammars will be used throughout this book to organize compiler front ends.

A grammar naturally describes the hierarchical structure of most programming language constructs. For example, an if-else statement in Java can have the form

**if** ( expression ) statement **else** statement

That is, an if-else statement is the concatenation of the keyword **if**, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword **else**, and another statement. Using the variable *expr* to denote an expression and the variable *stmt* to denote a statement, this structuring rule can be expressed as

*stmt* → **if** ( *expr* ) *stmt* **else** *stmt*

in which the arrow may be read as "can have the form." Such a rule is called a *production*. In a production, lexical elements like the keyword **if** and the parentheses are called *terminals*. Variables like *expr* and *stmt* represent sequences of terminals and are called *nonterminals*.

### 2.2.1 Definition of Grammars

A *context-free grammar* has four components:

1. A set of *terminal* symbols, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar.

2. A set of *nonterminals*, sometimes called "syntactic variables." Each nonterminal represents a set of strings of terminals, in a manner we shall describe.

3. A set of *productions*, where each production consists of a nonterminal, called the *head* or *left side* of the production, an arrow, and a sequence of

### Tokens Versus Terminals

In a compiler, the lexical analyzer reads the characters of the source program, groups them into lexically meaningful units called lexemes, and produces as output tokens representing these lexemes. A token consists of two components, a token name and an attribute value. The token names are abstract symbols that are used by the parser for syntax analysis. Often, we shall call these token names *terminals*, since they appear as terminal symbols in the grammar for a programming language. The attribute value, if present, is a pointer to the symbol table that contains additional information about the token. This additional information is not part of the grammar, so in our discussion of syntax analysis, often we refer to tokens and terminals synonymously.

terminals and/or nonterminals, called the *body* or *right side* of the production. The intuitive intent of a production is to specify one of the written forms of a construct; if the head nonterminal represents a construct, then the body represents a written form of the construct.

4. A designation of one of the nonterminals as the *start* symbol.

We specify grammars by listing their productions, with the productions for the start symbol listed first. We assume that digits, signs such as `<` and `<=`, and boldface strings such as **while** are terminals. An italicized name is a nonterminal, and any nonitalicized name or symbol may be assumed to be a terminal.[1] For notational convenience, productions with the same nonterminal as the head can have their bodies grouped, with the alternative bodies separated by the symbol |, which we read as "or."

**Example 2.1:** Several examples in this chapter use expressions consisting of digits and plus and minus signs; e.g., strings such as `9-5+2`, `3-1`, or `7`. Since a plus or minus sign must appear between two digits, we refer to such expressions as "lists of digits separated by plus or minus signs." The following grammar describes the syntax of these expressions. The productions are:

$$\textit{list} \rightarrow \textit{list}\ \texttt{+}\ \textit{digit} \tag{2.1}$$

$$\textit{list} \rightarrow \textit{list}\ \texttt{-}\ \textit{digit} \tag{2.2}$$

$$\textit{list} \rightarrow \textit{digit} \tag{2.3}$$

$$\textit{digit} \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \tag{2.4}$$

[1] Individual italic letters will be used for additional purposes, especially when grammars are studied in detail in Chapter 4. For example, we shall use $X$, $Y$, and $Z$ to talk about a symbol that is either a terminal or a nonterminal. However, any italicized name containing two or more characters will continue to represent a nonterminal.

The bodies of the three productions with nonterminal *list* as head equivalently can be grouped:

$$list \rightarrow list \texttt{ + } digit \mid list \texttt{ - } digit \mid digit$$

According to our conventions, the terminals of the grammar are the symbols

$$\texttt{+ - 0 1 2 3 4 5 6 7 8 9}$$

The nonterminals are the italicized names *list* and *digit*, with *list* being the start symbol because its productions are given first. □

We say a production is *for* a nonterminal if the nonterminal is the head of the production. A string of terminals is a sequence of zero or more terminals. The string of zero terminals, written as $\epsilon$, is called the *empty* string.[2]

### 2.2.2 Derivations

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal. The terminal strings that can be derived from the start symbol form the *language* defined by the grammar.

**Example 2.2:** The language defined by the grammar of Example 2.1 consists of lists of digits separated by plus and minus signs. The ten productions for the nonterminal *digit* allow it to stand for any of the terminals $\texttt{0}, \texttt{1}, \ldots, \texttt{9}$. From production (2.3), a single digit by itself is a list. Productions (2.1) and (2.2) express the rule that any list followed by a plus or minus sign and then another digit makes up a new list.

Productions (2.1) to (2.4) are all we need to define the desired language. For example, we can deduce that `9-5+2` is a *list* as follows.

a) `9` is a *list* by production (2.3), since `9` is a *digit*.

b) `9-5` is a *list* by production (2.2), since `9` is a *list* and `5` is a *digit*.

c) `9-5+2` is a *list* by production (2.1), since `9-5` is a *list* and `2` is a *digit*.

□

**Example 2.3:** A somewhat different sort of list is the list of parameters in a function call. In Java, the parameters are enclosed within parentheses, as in the call `max(x,y)` of function `max` with parameters `x` and `y`. One nuance of such lists is that an empty list of parameters may be found between the terminals `(` and `)`. We may start to develop a grammar for such sequences with the productions:

[2] Technically, $\epsilon$ can be a string of zero symbols from any alphabet (collection of symbols).

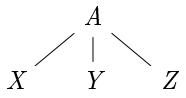
$$\begin{array}{rcl} \textit{call} & \rightarrow & \textbf{id}\ \texttt{(}\ \textit{optparams}\ \texttt{)} \\ \textit{optparams} & \rightarrow & \textit{params} \mid \epsilon \\ \textit{params} & \rightarrow & \textit{params}\ \texttt{,}\ \textit{param} \mid \textit{param} \end{array}$$

Note that the second possible body for *optparams* ("optional parameter list") is $\epsilon$, which stands for the empty string of symbols. That is, *optparams* can be replaced by the empty string, so a *call* can consist of a function name followed by the two-terminal string `()`. Notice that the productions for *params* are analogous to those for *list* in Example 2.1, with comma in place of the arithmetic operator `+` or `-`, and *param* in place of *digit*. We have not shown the productions for *param*, since parameters are really arbitrary expressions. Shortly, we shall discuss the appropriate productions for the various language constructs, such as expressions, statements, and so on. □

*Parsing* is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived from the start symbol of the grammar, then reporting syntax errors within the string. Parsing is one of the most fundamental problems in all of compiling; the main approaches to parsing are discussed in Chapter 4. In this chapter, for simplicity, we begin with source programs like `9-5+2` in which each character is a terminal; in general, a source program has multicharacter lexemes that are grouped by the lexical analyzer into tokens, whose first components are the terminals processed by the parser.

### 2.2.3 Parse Trees

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal $A$ has a production $A \rightarrow XYZ$, then a parse tree may have an interior node labeled $A$ with three children labeled $X$, $Y$, and $Z$, from left to right:




Formally, given a context-free grammar, a *parse tree* according to the grammar is a tree with the following properties:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by $\epsilon$.
3. Each interior node is labeled by a nonterminal.
4. If $A$ is the nonterminal labeling some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \cdots X_n$. Here, $X_1, X_2, \ldots, X_n$ each stand

### Tree Terminology

Tree data structures figure prominently in compiling.

- A tree consists of one or more *nodes*. Nodes may have *labels*, which in this book typically will be grammar symbols. When we draw a tree, we often represent the nodes by these labels only.
- Exactly one node is the *root*. All nodes except the root have a unique *parent*; the root has no parent. When we draw trees, we place the parent of a node above that node and draw an edge between them. The root is then the highest (top) node.
- If node $N$ is the parent of node $M$, then $M$ is a *child* of $N$. The children of one node are called *siblings*. They have an order, *from the left*, and when we draw trees, we order the childen of a given node in this manner.
- A node with no children is called a *leaf*. Other nodes — those with one or more children — are *interior nodes*.
- A *descendant* of a node $N$ is either $N$ itself, a child of $N$, a child of a child of $N$, and so on, for any number of levels. We say node $N$ is an *ancestor* of node $M$ if $M$ is a descendant of $N$.

for a symbol that is either a terminal or a nonterminal. As a special case, if $A \rightarrow \epsilon$ is a production, then a node labeled $A$ may have a single child labeled $\epsilon$.

**Example 2.4:** The derivation of `9-5+2` in Example 2.2 is illustrated by the tree in Fig. 2.5. Each node in the tree is labeled by a grammar symbol. An interior node and its children correspond to a production; the interior node corresponds to the head of the production, the children to the body.

In Fig. 2.5, the root is labeled *list*, the start symbol of the grammar in Example 2.1. The children of the root are labeled, from left to right, *list*, `+`, and *digit*. Note that

$$list \rightarrow list \texttt{ + } digit$$

is a production in the grammar of Example 2.1. The left child of the root is similar to the root, with a child labeled `-` instead of `+`. The three nodes labeled *digit* each have one child that is labeled by a digit. □

From left to right, the leaves of a parse tree form the *yield* of the tree, which is the string *generated* or *derived* from the nonterminal at the root of the parse

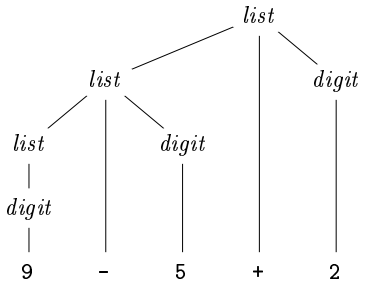



Figure 2.5: Parse tree for 9-5+2 according to the grammar in Example 2.1

tree. In Fig. 2.5, the yield is 9-5+2; for convenience, all the leaves are shown at the bottom level. Henceforth, we shall not necessarily line up the leaves in this way. Any tree imparts a natural left-to-right order to its leaves, based on the idea that if $X$ and $Y$ are two children with the same parent, and $X$ is to the left of $Y$, then all descendants of $X$ are to the left of descendants of $Y$.

Another definition of the language generated by a grammar is as the set of strings that can be generated by some parse tree. The process of finding a parse tree for a given string of terminals is called *parsing* that string.

### 2.2.4 Ambiguity

We have to be careful in talking about *the* structure of a string according to a grammar. A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be *ambiguous*. To show that a grammar is ambiguous, all we need to do is find a terminal string that is the yield of more than one parse tree. Since a string with more than one parse tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities.

**Example 2.5:** Suppose we used a single nonterminal *string* and did not distinguish between digits and lists, as in Example 2.1. We could have written the grammar

*string* → *string* + *string* | *string* - *string* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Merging the notion of *digit* and *list* into the nonterminal *string* makes superficial sense, because a single *digit* is a special case of a *list*.

However, Fig. 2.6 shows that an expression like 9-5+2 has more than one parse tree with this grammar. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and 9-(5+2). This second parenthesization gives the expression the unexpected value 2 rather than the customary value 6. The grammar of Example 2.1 does not permit this interpretation. □

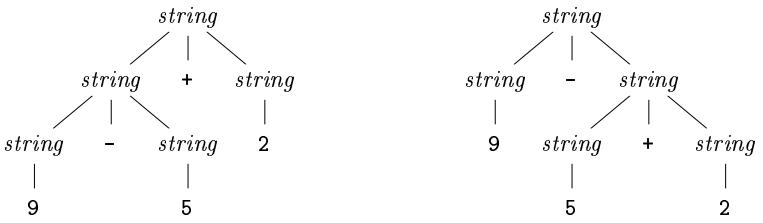



Figure 2.6: Two parse trees for 9-5+2

### 2.2.5 Associativity of Operators

By convention, 9+5+2 is equivalent to (9+5)+2 and 9-5-2 is equivalent to (9-5)-2. When an operand like 5 has operators to its left and right, conventions are needed for deciding which operator applies to that operand. We say that the operator + *associates* to the left, because an operand with plus signs on both sides of it belongs to the operator to its left. In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left-associative.

Some common operators such as exponentiation are right-associative. As another example, the assignment operator = in C and its descendants is right-associative; that is, the expression a=b=c is treated in the same way as the expression a=(b=c).

Strings like a=b=c with a right-associative operator are generated by the following grammar:

$$\begin{array}{lcl} \textit{right} & \rightarrow & \textit{letter} = \textit{right} \mid \textit{letter} \\ \textit{letter} & \rightarrow & \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \end{array}$$

The contrast between a parse tree for a left-associative operator like - and a parse tree for a right-associative operator like = is shown by Fig. 2.7. Note that the parse tree for 9-5-2 grows down towards the left, whereas the parse tree for a=b=c grows down towards the right.

### 2.2.6 Precedence of Operators

Consider the expression 9+5*2. There are two possible interpretations of this expression: (9+5)*2 or 9+(5*2). The associativity rules for + and * apply to occurrences of the same operator, so they do not resolve this ambiguity. Rules defining the relative precedence of operators are needed when more than one kind of operator is present.

We say that * has *higher precedence* than + if * takes its operands before + does. In ordinary arithmetic, multiplication and division have higher precedence than addition and subtraction. Therefore, 5 is taken by * in both 9+5*2 and 9*5+2; i.e., the expressions are equivalent to 9+(5*2) and (9*5)+2, respectively.

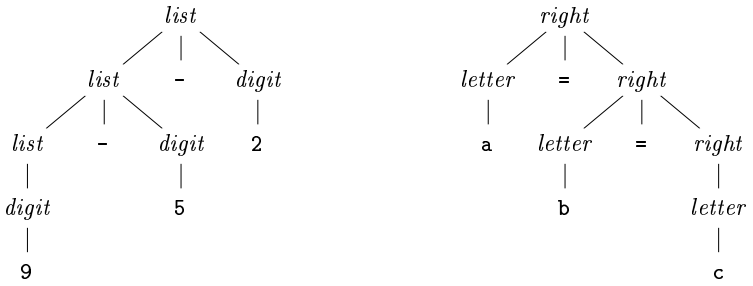



Figure 2.7: Parse trees for left- and right-associative grammars

**Example 2.6:** A grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators. We start with the four common arithmetic operators and a precedence table, showing the operators in order of increasing precedence. Operators on the same line have the same associativity and precedence:

| | |
|---|---|
| left-associative: | `+ -` |
| left-associative: | `* /` |

We create two nonterminals *expr* and *term* for the two levels of precedence, and an extra nonterminal *factor* for generating basic units in expressions. The basic units in expressions are presently digits and parenthesized expressions.

$$factor \rightarrow \textbf{digit} \mid \texttt{(}\ expr\ \texttt{)}$$

Now consider the binary operators, `*` and `/`, that have the highest precedence. Since these operators associate to the left, the productions are similar to those for lists that associate to the left.

$$\begin{array}{rcl} term & \rightarrow & term\ \texttt{*}\ factor \\ & \mid & term\ \texttt{/}\ factor \\ & \mid & factor \end{array}$$

Similarly, *expr* generates lists of terms separated by the additive operators.

$$\begin{array}{rcl} expr & \rightarrow & expr\ \texttt{+}\ term \\ & \mid & expr\ \texttt{-}\ term \\ & \mid & term \end{array}$$

The resulting grammar is therefore

$$\begin{array}{rcl} expr & \rightarrow & expr\ \texttt{+}\ term \mid expr\ \texttt{-}\ term \mid term \\ term & \rightarrow & term\ \texttt{*}\ factor \mid term\ \texttt{/}\ factor \mid factor \\ factor & \rightarrow & \textbf{digit} \mid \texttt{(}\ expr\ \texttt{)} \end{array}$$

### Generalizing the Expression Grammar of Example 2.6

We can think of a factor as an expression that cannot be "torn apart" by any operator. By "torn apart," we mean that placing an operator next to any factor, on either side, does not cause any piece of the factor, other than the whole, to become an operand of that operator. If the factor is a parenthesized expression, the parentheses protect against such "tearing," while if the factor is a single operand, it cannot be torn apart.

A term (that is not also a factor) is an expression that can be torn apart by operators of the highest precedence: `*` and `/`, but not by the lower-precedence operators. An expression (that is not a term or factor) can be torn apart by any operator.

We can generalize this idea to any number $n$ of precedence levels. We need $n+1$ nonterminals. The first, like *factor* in Example 2.6, can never be torn apart. Typically, the production bodies for this nonterminal are only single operands and parenthesized expressions. Then, for each precedence level, there is one nonterminal representing expressions that can be torn apart only by operators at that level or higher. Typically, the productions for this nonterminal have bodies representing uses of the operators at that level, plus one body that is just the nonterminal for the next higher level.

With this grammar, an expression is a list of terms separated by either `+` or `-` signs, and a term is a list of factors separated by `*` or `/` signs. Notice that any parenthesized expression is a factor, so with parentheses we can develop expressions that have arbitrarily deep nesting (and arbitrarily deep trees). □

**Example 2.7:** Keywords allow us to recognize statements, since most statement begin with a keyword or a special character. Exceptions to this rule include assignments and procedure calls. The statements defined by the (ambiguous) grammar in Fig. 2.8 are legal in Java.

In the first production for *stmt*, the terminal **id** represents any identifier. The productions for *expression* are not shown. The assignment statements specified by the first production are legal in Java, although Java treats `=` as an assignment operator that can appear within an expression. For example, Java allows `a = b = c`, which this grammar does not.

The nonterminal *stmts* generates a possibly empty list of statements. The second production for *stmts* generates the empty list $\epsilon$. The first production generates a possibly empty list of statements followed by a statement.

The placement of semicolons is subtle; they appear at the end of every body that does not end in *stmt*. This approach prevents the build-up of semicolons after statements such as if- and while-, which end with nested substatements. When the nested substatement is an assignment or a do-while, a semicolon will be generated as part of the substatement. □

$$
\begin{array}{lcl}
\textit{stmt} & \rightarrow & \textbf{id} = \textit{expression} \ ; \\
& | & \textbf{if} \ ( \ \textit{expression} \ ) \ \textit{stmt} \\
& | & \textbf{if} \ ( \ \textit{expression} \ ) \ \textit{stmt} \ \textbf{else} \ \textit{stmt} \\
& | & \textbf{while} \ ( \ \textit{expression} \ ) \ \textit{stmt} \\
& | & \textbf{do} \ \textit{stmt} \ \textbf{while} \ ( \ \textit{expression} \ ) \ ; \\
& | & \{ \ \textit{stmts} \ \} \\
\\
\textit{stmts} & \rightarrow & \textit{stmts} \ \textit{stmt} \\
& | & \epsilon
\end{array}
$$

Figure 2.8: A grammar for a subset of Java statements

### 2.2.7 Exercises for Section 2.2

**Exercise 2.2.1:** Consider the context-free grammar

$$S \ \rightarrow \ S \ S \ \texttt{+} \ | \ S \ S \ \texttt{*} \ | \ \mathbf{a}$$

a) Show how the string `aa+a*` can be generated by this grammar.

b) Construct a parse tree for this string.

c) What language does this grammar generate? Justify your answer.

**Exercise 2.2.2:** What language is generated by the following grammars? In each case justify your answer.

a) $S \ \rightarrow \ 0 \ S \ 1 \ | \ 0 \ 1$

b) $S \ \rightarrow \ \texttt{+} \ S \ S \ | \ \texttt{-} \ S \ S \ | \ \mathbf{a}$

c) $S \ \rightarrow \ S \ ( \ S \ ) \ S \ | \ \epsilon$

d) $S \ \rightarrow \ \mathbf{a} \ S \ \mathbf{b} \ S \ | \ \mathbf{b} \ S \ \mathbf{a} \ S \ | \ \epsilon$

e) $S \ \rightarrow \ \mathbf{a} \ | \ S \ \texttt{+} \ S \ | \ S \ S \ | \ S \ \texttt{*} \ | \ ( \ S \ )$

**Exercise 2.2.3:** Which of the grammars in Exercise 2.2.2 are ambiguous?

**Exercise 2.2.4:** Construct unambiguous context-free grammars for each of the following languages. In each case show that your grammar is correct.

a) Arithmetic expressions in postfix notation.

b) Left-associative lists of identifiers separated by commas.

c) Right-associative lists of identifiers separated by commas.

d) Arithmetic expressions of integers and identifiers with the four binary operators `+`, `-`, `*`, `/`.

! e) Add unary plus and minus to the arithmetic operators of (d).

**Exercise 2.2.5:**

a) Show that all binary strings generated by the following grammar have values divisible by 3. *Hint.* Use induction on the number of nodes in a parse tree.

$$num \rightarrow 11 \mid 1001 \mid num\ 0 \mid num\ num$$

b) Does the grammar generate all binary strings with values divisible by 3?

**Exercise 2.2.6:** Construct a context-free grammar for roman numerals.

## 2.3 Syntax-Directed Translation

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar. For example, consider an expression *expr* generated by the production

$$expr \rightarrow expr_1 + term$$

Here, *expr* is the sum of the two subexpressions $expr_1$ and *term*. (The subscript in $expr_1$ is used only to distinguish the instance of *expr* in the production body from the head of the production). We can translate *expr* by exploiting its structure, as in the following pseudo-code:

translate $expr_1$;
translate *term*;
handle +;

Using a variant of this pseudocode, we shall build a syntax tree for *expr* in Section 2.8 by building syntax trees for $expr_1$ and *term* and then handling + by constructing a node for it. For convenience, the example in this section is the translation of infix expressions into postfix notation.

This section introduces two concepts related to syntax-directed translation:

- *Attributes.* An *attribute* is any quantity associated with a programming construct. Examples of attributes are data types of expressions, the number of instructions in the generated code, or the location of the first instruction in the generated code for a construct, among many other possibilities. Since we use grammar symbols (nonterminals and terminals) to represent programming constructs, we extend the notion of attributes from constructs to the symbols that represent them.

- (*Syntax-directed*) *translation schemes.* A *translation scheme* is a notation for attaching program fragments to the productions of a grammar. The program fragments are executed when the production is used during syntax analysis. The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied.

Syntax-directed translations will be used throughout this chapter to translate infix expressions into postfix notation, to evaluate expressions, and to build syntax trees for programming constructs. A more detailed discussion of syntax-directed formalisms appears in Chapter 5.

### 2.3.1 Postfix Notation

The examples in this section deal with translation into postfix notation. The *postfix notation* for an expression $E$ can be defined inductively as follows:

1. If $E$ is a variable or constant, then the postfix notation for $E$ is $E$ itself.
2. If $E$ is an expression of the form $E_1$ **op** $E_2$, where **op** is any binary operator, then the postfix notation for $E$ is $E_1'$ $E_2'$ **op**, where $E_1'$ and $E_2'$ are the postfix notations for $E_1$ and $E_2$, respectively.
3. If $E$ is a parenthesized expression of the form $(E_1)$, then the postfix notation for $E$ is the same as the postfix notation for $E_1$.

**Example 2.8:** The postfix notation for `(9-5)+2` is `95-2+`. That is, the translations of `9`, `5`, and `2` are the constants themselves, by rule (1). Then, the translation of `9-5` is `95-` by rule (2). The translation of `(9-5)` is the same by rule (3). Having translated the parenthesized subexpression, we may apply rule (2) to the entire expression, with `(9-5)` in the role of $E_1$ and `2` in the role of $E_2$, to get the result `95-2+`.

As another example, the postfix notation for `9-(5+2)` is `952+-`. That is, `5+2` is first translated into `52+`, and this expression becomes the second argument of the minus sign. □

No parentheses are needed in postfix notation, because the position and *arity* (number of arguments) of the operators permits only one decoding of a postfix expression. The "trick" is to repeatedly scan the postfix string from the left, until you find an operator. Then, look to the left for the proper number of operands, and group this operator with its operands. Evaluate the operator on the operands, and replace them by the result. Then repeat the process, continuing to the right and searching for another operator.

**Example 2.9:** Consider the postfix expression `952+-3*`. Scanning from the left, we first encounter the plus sign. Looking to its left we find operands `5` and `2`. Their sum, `7`, replaces `52+`, and we have the string `97-3*`. Now, the leftmost

operator is the minus sign, and its operands are 9 and 7. Replacing these by the result of the subtraction leaves 23*. Last, the multiplication sign applies to 2 and 3, giving the result 6. □

### 2.3.2 Synthesized Attributes

The idea of associating quantities with programming constructs—for example, values and types with expressions—can be expressed in terms of grammars. We associate attributes with nonterminals and terminals. Then, we attach rules to the productions of the grammar; these rules describe how the attributes are computed at those nodes of the parse tree where the production in question is used to relate a node to its children.

A *syntax-directed definition* associates

1. With each grammar symbol, a set of attributes, and
2. With each production, a set of *semantic rules* for computing the values of the attributes associated with the symbols appearing in the production.

Attributes can be evaluated as follows. For a given input string $x$, construct a parse tree for $x$. Then, apply the semantic rules to evaluate attributes at each node in the parse tree, as follows.

Suppose a node $N$ in a parse tree is labeled by the grammar symbol $X$. We write $X.a$ to denote the value of attribute $a$ of $X$ at that node. A parse tree showing the attribute values at each node is called an *annotated* parse tree. For example, Fig. 2.9 shows an annotated parse tree for 9-5+2 with an attribute $t$ associated with the nonterminals *expr* and *term*. The value 95-2+ of the attribute at the root is the postfix notation for 9-5+2. We shall see shortly how these expressions are computed.

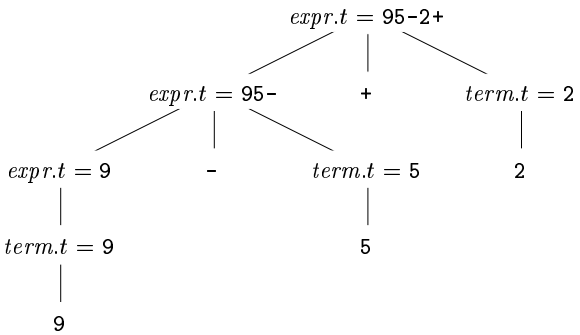



Figure 2.9: Attribute values at nodes in a parse tree

An attribute is said to be *synthesized* if its value at a parse-tree node $N$ is determined from attribute values at the children of $N$ and at $N$ itself. Synthesized

attributes have the desirable property that they can be evaluated during a single bottom-up traversal of a parse tree. In Section 5.1.1 we shall discuss another important kind of attribute: the "inherited" attribute. Informally, inherited attributes have their value at a parse-tree node determined from attribute values at the node itself, its parent, and its siblings in the parse tree.

**Example 2.10 :** The annotated parse tree in Fig. 2.9 is based on the syntax-directed definition in Fig. 2.10 for translating expressions consisting of digits separated by plus or minus signs into postfix notation. Each nonterminal has a string-valued attribute $t$ that represents the postfix notation for the expression generated by that nonterminal in a parse tree. The symbol || in the semantic rule is the operator for string concatenation.

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $expr \rightarrow expr_1$ + $term$ | $expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$ |
| $expr \rightarrow expr_1$ - $term$ | $expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$ |
| $expr \rightarrow term$ | $expr.t = term.t$ |
| $term \rightarrow$ 0 | $term.t = '0'$ |
| $term \rightarrow$ 1 | $term.t = '1'$ |
| ... | ... |
| $term \rightarrow$ 9 | $term.t = '9'$ |

Figure 2.10: Syntax-directed definition for infix to postfix translation

The postfix form of a digit is the digit itself; e.g., the semantic rule associated with the production $term \rightarrow$ 9 defines $term.t$ to be 9 itself whenever this production is used at a node in a parse tree. The other digits are translated similarly. As another example, when the production $expr \rightarrow term$ is applied, the value of $term.t$ becomes the value of $expr.t$.

The production $expr \rightarrow expr_1$ + $term$ derives an expression containing a plus operator.[3] The left operand of the plus operator is given by $expr_1$ and the right operand by $term$. The semantic rule

$$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$$

associated with this production constructs the value of attribute $expr.t$ by concatenating the postfix forms $expr_1.t$ and $term.t$ of the left and right operands, respectively, and then appending the plus sign. This rule is a formalization of the definition of "postfix expression." □

[3]In this and many other rules, the same nonterminal (*expr*, here) appears several times. The purpose of the subscript 1 in $expr_1$ is to distinguish the two occurrences of *expr* in the production; the "1" is not part of the nonterminal. See the box on "Convention Distinguishing Uses of a Nonterminal" for more details.

### Convention Distinguishing Uses of a Nonterminal

In rules, we often have a need to distinguish among several uses of the same nonterminal in the head and/or body of a production; e.g., see Example 2.10. The reason is that in the parse tree, different nodes labeled by the same nonterminal usually have different values for their translations. We shall adopt the following convention: the nonterminal appears unsubscripted in the head and with distinct subscripts in the body. These are all occurrences of the same nonterminal, and the subscript is not part of its name. However, the reader should be alert to the difference between examples of specific translations, where this convention is used, and generic productions like $A \rightarrow X_1 X_2, \ldots, X_n$, where the subscripted $X$'s represent an arbitrary list of grammar symbols, and are *not* instances of one particular nonterminal called $X$.

### 2.3.3 Simple Syntax-Directed Definitions

The syntax-directed definition in Example 2.10 has the following important property: the string representing the translation of the nonterminal at the head of each production is the concatenation of the translations of the nonterminals in the production body, in the same order as in the production, with some optional additional strings interleaved. A syntax-directed definition with this property is termed *simple*.

**Example 2.11 :** Consider the first production and semantic rule from Fig. 2.10:

| PRODUCTION | SEMANTIC RULE | (2.5) |
|---|---|---|
| $expr \rightarrow expr_1 + term$ | $expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$ | |

Here the translation $expr.t$ is the concatenation of the translations of $expr_1$ and $term$, followed by the symbol +. Notice that $expr_1$ and $term$ appear in the same order in both the production body and the semantic rule. There are no additional symbols before or between their translations. In this example, the only extra symbol occurs at the end. □

When translation schemes are discussed, we shall see that a simple syntax-directed definition can be implemented by printing only the additional strings, in the order they appear in the definition.

### 2.3.4 Tree Traversals

Tree traversals will be used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme. A *traversal* of a tree starts at the root and visits each node of the tree in some order.

A *depth-first* traversal starts at the root and recursively visits the children of each node in any order, not necessarily from left to right. It is called "depth-first" because it visits an unvisited child of a node whenever it can, so it visits nodes as far away from the root (as "deep") as quickly as it can.

The procedure $visit(N)$ in Fig. 2.11 is a depth first traversal that visits the children of a node in left-to-right order, as shown in Fig. 2.12. In this traversal, we have included the action of evaluating translations at each node, just before we finish with the node (that is, after translations at the children have surely been computed). In general, the actions associated with a traversal can be whatever we choose, or nothing at all.

```
procedure visit(node N) {
        for ( each child C of N, from left to right ) {
                visit(C);
        }
        evaluate semantic rules at node N;
}
```

Figure 2.11: A depth-first traversal of a tree

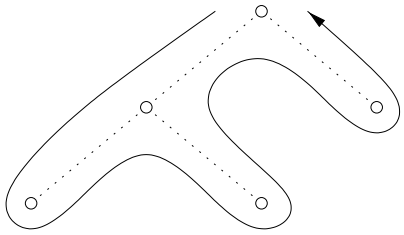


Figure 2.12: Example of a depth-first traversal of a tree

A syntax-directed definition does not impose any specific order for the evaluation of attributes on a parse tree; any evaluation order that computes an attribute $a$ after all the other attributes that $a$ depends on is acceptable. Synthesized attributes can be evaluated during any *bottom-up* traversal, that is, a traversal that evaluates attributes at a node after having evaluated attributes at its children. In general, with both synthesized and inherited attributes, the matter of evaluation order is quite complex; see Section 5.2.

### 2.3.5 Translation Schemes

The syntax-directed definition in Fig. 2.10 builds up a translation by attaching strings as attributes to the nodes in the parse tree. We now consider an alternative approach that does not need to manipulate strings; it produces the same translation incrementally, by executing program fragments.

### Preorder and Postorder Traversals

Preorder and postorder traversals are two important special cases of depth-first traversals in which we visit the children of each node from left to right.

Often, we traverse a tree to perform some particular action at each node. If the action is done when we first visit a node, then we may refer to the traversal as a *preorder traversal*. Similarly, if the action is done just before we leave a node for the last time, then we say it is a *postorder traversal* of the tree. The procedure $visit(N)$ in Fig. 2.11 is an example of a postorder traversal.

Preorder and postorder traversals define corresponding orderings on nodes, based on when the action at a node would be performed. The *preorder* of a (sub)tree rooted at node $N$ consists of $N$, followed by the preorders of the subtrees of each of its children, if any, from the left. The *postorder* of a (sub)tree rooted at $N$ consists of the postorders of each of the subtrees for the children of $N$, if any, from the left, followed by $N$ itself.

A syntax-directed translation scheme is a notation for specifying a translation by attaching program fragments to productions in a grammar. A translation scheme is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly specified.

Program fragments embedded within production bodies are called *semantic actions*. The position at which an action is to be executed is shown by enclosing it between curly braces and writing it within the production body, as in

$$rest \rightarrow + \; term \; \{\text{print}('+')\} \; rest_1$$

We shall see such rules when we consider an alternative form of grammar for expressions, where the nonterminal *rest* represents "everything but the first term of an expression." This form of grammar is discussed in Section 2.4.5. Again, the subscript in $rest_1$ distinguishes this instance of nonterminal *rest* in the production body from the instance of *rest* at the head of the production.

When drawing a parse tree for a translation scheme, we indicate an action by constructing an extra child for it, connected by a dashed line to the node that corresponds to the head of the production. For example, the portion of the parse tree for the above production and action is shown in Fig. 2.13. The node for a semantic action has no children, so the action is performed when that node is first seen.

**Example 2.12:** The parse tree in Fig. 2.14 has print statements at extra leaves, which are attached by dashed lines to interior nodes of the parse tree. The translation scheme appears in Fig. 2.15. The underlying grammar generates expressions consisting of digits separated by plus and minus signs. The

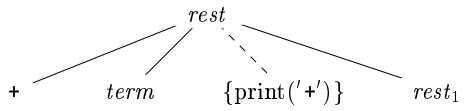



Figure 2.13: An extra leaf is constructed for a semantic action

actions embedded in the production bodies translate such expressions into postfix notation, provided we perform a left-to-right depth-first traversal of the tree and execute each print statement when we visit its leaf.

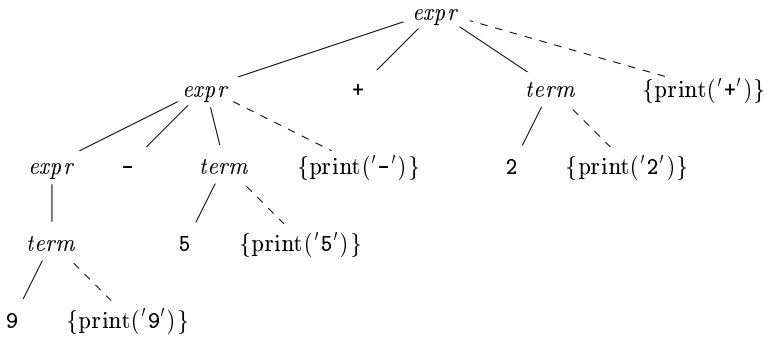



Figure 2.14: Actions translating 9-5+2 into 95-2+

| | | | |
|---|---|---|---|
| *expr* | → | $expr_1$ + *term* | {print('+')} |
| *expr* | → | $expr_1$ - *term* | {print('-')} |
| *expr* | → | *term* | |
| *term* | → | 0 | {print('0')} |
| *term* | → | 1 | {print('1')} |
| | | ... | |
| *term* | → | 9 | {print('9')} |

Figure 2.15: Actions for translating into postfix notation

The root of Fig. 2.14 represents the first production in Fig. 2.15. In a postorder traversal, we first perform all the actions in the leftmost subtree of the root, for the left operand, also labeled *expr* like the root. We then visit the leaf + at which there is no action. We next perform the actions in the subtree for the right operand *term* and, finally, the semantic action { print('+') } at the extra node.

Since the productions for *term* have only a digit on the right side, that digit is printed by the actions for the productions. No output is necessary for the production *expr* → *term*, and only the operator needs to be printed in the

action for each of the first two productions. When executed during a postorder traversal of the parse tree, the actions in Fig. 2.14 print `95-2+`. □

Note that although the schemes in Fig. 2.10 and Fig. 2.15 produce the same translation, they construct it differently; Fig. 2.10 attaches strings as attributes to the nodes in the parse tree, while the scheme in Fig. 2.15 prints the translation incrementally, through semantic actions.

The semantic actions in the parse tree in Fig. 2.14 translate the infix expression `9-5+2` into `95-2+` by printing each character in `9-5+2` exactly once, without using any storage for the translation of subexpressions. When the output is created incrementally in this fashion, the order in which the characters are printed is significant.

The implementation of a translation scheme must ensure that semantic actions are performed in the order they would appear during a postorder traversal of a parse tree. The implementation need not actually construct a parse tree (often it does not), as long as it ensures that the semantic actions are performed as if we constructed a parse tree and then executed the actions during a postorder traversal.

### 2.3.6 Exercises for Section 2.3

**Exercise 2.3.1:** Construct a syntax-directed translation scheme that translates arithmetic expressions from infix notation into prefix notation in which an operator appears before its operands; e.g., $-xy$ is the prefix notation for $x - y$. Give annotated parse trees for the inputs `9-5+2` and `9-5*2`.

**Exercise 2.3.2:** Construct a syntax-directed translation scheme that translates arithmetic expressions from postfix notation into infix notation. Give annotated parse trees for the inputs `95-2*` and `952*-`.

**Exercise 2.3.3:** Construct a syntax-directed translation scheme that translates integers into roman numerals.

! **Exercise 2.3.4:** Construct a syntax-directed translation scheme that translates roman numerals up to 2000 into integers.

**Exercise 2.3.5:** Construct a syntax-directed translation scheme to translate postfix arithmetic expressions into equivalent prefix arithmetic expressions.

## 2.4 Parsing

Parsing is the process of determining how a string of terminals can be generated by a grammar. In discussing this problem, it is helpful to think of a parse tree being constructed, even though a compiler may not construct one, in practice. However, a parser must be capable of constructing the tree in principle, or else the translation cannot be guaranteed correct.

This section introduces a parsing method called "recursive descent," which can be used both to parse and to implement syntax-directed translators. A complete Java program, implementing the translation scheme of Fig. 2.15, appears in the next section. A viable alternative is to use a software tool to generate a translator directly from a translation scheme. Section 4.9 describes such a tool — Yacc; it can implement the translation scheme of Fig. 2.15 without modification.

For any context-free grammar there is a parser that takes at most $O(n^3)$ time to parse a string of $n$ terminals. But cubic time is generally too expensive. Fortunately, for real programming languages, we can generally design a grammar that can be parsed quickly. Linear-time algorithms suffice to parse essentially all languages that arise in practice. Programming-language parsers almost always make a single left-to-right scan over the input, looking ahead one terminal at a time, and constructing pieces of the parse tree as they go.

Most parsing methods fall into one of two classes, called the *top-down* and *bottom-up* methods. These terms refer to the order in which nodes in the parse tree are constructed. In top-down parsers, construction starts at the root and proceeds towards the leaves, while in bottom-up parsers, construction starts at the leaves and proceeds towards the root. The popularity of top-down parsers is due to the fact that efficient parsers can be constructed more easily by hand using top-down methods. Bottom-up parsing, however, can handle a larger class of grammars and translation schemes, so software tools for generating parsers directly from grammars often use bottom-up methods.

### 2.4.1 Top-Down Parsing

We introduce top-down parsing by considering a grammar that is well-suited for this class of methods. Later in this section, we consider the construction of top-down parsers in general. The grammar in Fig. 2.16 generates a subset of the statements of C or Java. We use the boldface terminals **if** and **for** for the keywords "`if`" and "`for`", respectively, to emphasize that these character sequences are treated as units, i.e., as single terminal symbols. Further, the terminal **expr** represents expressions; a more complete grammar would use a nonterminal *expr* and have productions for nonterminal *expr*. Similarly, **other** is a terminal representing other statement constructs.

The top-down construction of a parse tree like the one in Fig. 2.17, is done by starting with the root, labeled with the starting nonterminal *stmt*, and repeatedly performing the following two steps.

1. At node $N$, labeled with nonterminal $A$, select one of the productions for $A$ and construct children at $N$ for the symbols in the production body.

2. Find the next node at which a subtree is to be constructed, typically the leftmost unexpanded nonterminal of the tree.

For some grammars, the above steps can be implemented during a single left-to-right scan of the input string. The current terminal being scanned in the

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{expr}\ ; \\
 & | & \textbf{if}\ (\ \textbf{expr}\ )\ \mathit{stmt} \\
 & | & \textbf{for}\ (\ \mathit{optexpr}\ ;\ \mathit{optexpr}\ ;\ \mathit{optexpr}\ )\ \mathit{stmt} \\
 & | & \textbf{other} \\
 & & \\
\mathit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

Figure 2.16: A grammar for some statements in C and Java

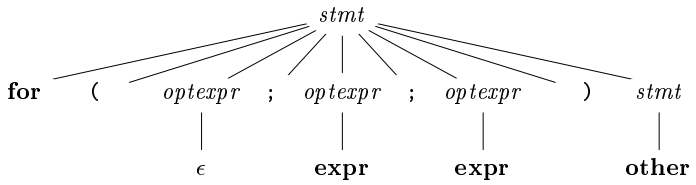



Figure 2.17: A parse tree according to the grammar in Fig. 2.16

input is frequently referred to as the *lookahead* symbol. Initially, the lookahead symbol is the first, i.e., leftmost, terminal of the input string. Figure 2.18 illustrates the construction of the parse tree in Fig. 2.17 for the input string

**for ( ; expr ; expr ) other**

Initially, the terminal **for** is the lookahead symbol, and the known part of the parse tree consists of the root, labeled with the starting nonterminal *stmt* in Fig. 2.18(a). The objective is to construct the remainder of the parse tree in such a way that the string generated by the parse tree matches the input string.

For a match to occur, the nonterminal *stmt* in Fig. 2.18(a) must derive a string that starts with the lookahead symbol **for**. In the grammar of Fig. 2.16, there is just one production for *stmt* that can derive such a string, so we select it, and construct the children of the root labeled with the symbols in the production body. This expansion of the parse tree is shown in Fig. 2.18(b).

Each of the three snapshots in Fig. 2.18 has arrows marking the lookahead symbol in the input and the node in the parse tree that is being considered. Once children are constructed at a node, we next consider the leftmost child. In Fig. 2.18(b), children have just been constructed at the root, and the leftmost child labeled with **for** is being considered.

When the node being considered in the parse tree is for a terminal, and the terminal matches the lookahead symbol, then we advance in both the parse tree and the input. The next terminal in the input becomes the new lookahead symbol, and the next child in the parse tree is considered. In Fig. 2.18(c), the arrow in the parse tree has advanced to the next child of the root, and the arrow

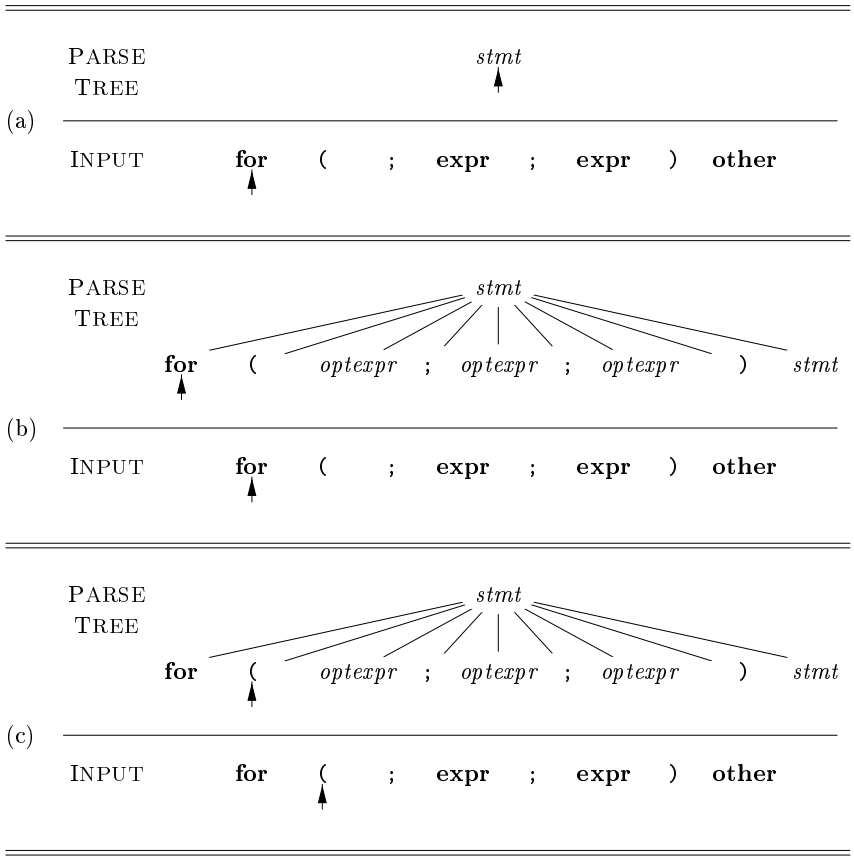



Figure 2.18: Top-down parsing while scanning the input from left to right

in the input has advanced to the next terminal, which is (. A further advance will take the arrow in the parse tree to the child labeled with nonterminal *optexpr* and take the arrow in the input to the terminal ;.

At the nonterminal node labeled *optexpr*, we repeat the process of selecting a production for a nonterminal. Productions with $\epsilon$ as the body ("$\epsilon$-productions") require special treatment. For the moment, we use them as a default when no other production can be used; we return to them in Section 2.4.3. With nonterminal *optexpr* and lookahead ;, the $\epsilon$-production is used, since ; does not match the only other production for *optexpr*, which has terminal **expr** as its body.

In general, the selection of a production for a nonterminal may involve trial-and-error; that is, we may have to try a production and backtrack to try another production if the first is found to be unsuitable. A production is unsuitable if, after using the production, we cannot complete the tree to match the input

string. Backtracking is not needed, however, in an important special case called predictive parsing, which we discuss next.

### 2.4.2 Predictive Parsing

*Recursive-descent parsing* is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input. One procedure is associated with each nonterminal of a grammar. Here, we consider a simple form of recursive-descent parsing, called *predictive parsing*, in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal. The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

The predictive parser in Fig. 2.19 consists of procedures for the nonterminals *stmt* and *optexpr* of the grammar in Fig. 2.16 and an additional procedure *match*, used to simplify the code for *stmt* and *optexpr*. Procedure $match(t)$ compares its argument $t$ with the lookahead symbol and advances to the next input terminal if they match. Thus *match* changes the value of variable *lookahead*, a global variable that holds the currently scanned input terminal.

Parsing begins with a call of the procedure for the starting nonterminal *stmt*. With the same input as in Fig. 2.18, *lookahead* is initially the first terminal **for**. Procedure *stmt* executes code corresponding to the production

$$stmt \rightarrow \textbf{for}\ (\ optexpr\ ;\ optexpr\ ;\ optexpr\ )\ stmt$$

In the code for the production body — that is, the **for** case of procedure *stmt* — each terminal is matched with the lookahead symbol, and each nonterminal leads to a call of its procedure, in the following sequence of calls:

$match(\textbf{for});\ \ match('(');$
$optexpr();\ \ match(';');\ \ optexpr();\ \ match(';');\ \ optexpr();$
$match(')');\ \ stmt();$

Predictive parsing relies on information about the first symbols that can be generated by a production body. More precisely, let $\alpha$ be a string of grammar symbols (terminals and/or nonterminals). We define FIRST($\alpha$) to be the set of terminals that appear as the first symbols of one or more strings of terminals generated from $\alpha$. If $\alpha$ is $\epsilon$ or can generate $\epsilon$, then $\epsilon$ is also in FIRST($\alpha$).

The details of how one computes FIRST($\alpha$) are in Section 4.4.2. Here, we shall just use ad hoc reasoning to deduce the symbols in FIRST($\alpha$); typically, $\alpha$ will either begin with a terminal, which is therefore the only symbol in FIRST($\alpha$), or $\alpha$ will begin with a nonterminal whose production bodies begin with terminals, in which case these terminals are the only members of FIRST($\alpha$).

For example, with respect to the grammar of Fig. 2.16, the following are correct calculations of FIRST.

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr); match(';'); break;
        case if:
                match(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

Figure 2.19: Pseudocode for a predictive parser

$$\begin{aligned} \text{FIRST}(\textit{stmt}) &= \{\textbf{expr}, \textbf{if}, \textbf{for}, \textbf{other}\} \\ \text{FIRST}(\textbf{expr ;}) &= \{\textbf{expr}\} \end{aligned}$$

The FIRST sets must be considered if there are two productions $A \to \alpha$ and $A \to \beta$. Ignoring $\epsilon$-productions for the moment, predictive parsing requires FIRST($\alpha$) and FIRST($\beta$) to be disjoint. The lookahead symbol can then be used to decide which production to use; if the lookahead symbol is in FIRST($\alpha$), then $\alpha$ is used. Otherwise, if the lookahead symbol is in FIRST($\beta$), then $\beta$ is used.

### 2.4.3 When to Use $\epsilon$-Productions

Our predictive parser uses an $\epsilon$-production as a default when no other production can be used. With the input of Fig. 2.18, after the terminals **for** and ( are matched, the lookahead symbol is ;. At this point procedure *optexpr* is called, and the code

if ( *lookahead* == **expr** ) *match*(**expr**);

in its body is executed. Nonterminal *optexpr* has two productions, with bodies **expr** and $\epsilon$. The lookahead symbol ";" does not match the terminal **expr**, so the production with body **expr** cannot apply. In fact, the procedure returns without changing the lookahead symbol or doing anything else. Doing nothing corresponds to applying an $\epsilon$-production.

More generally, consider a variant of the productions in Fig. 2.16 where *optexpr* generates an expression nonterminal instead of the terminal **expr**:

$$\begin{array}{lcl} \mathit{optexpr} & \rightarrow & \mathit{expr} \\ & | & \epsilon \end{array}$$

Thus, *optexpr* either generates an expression using nonterminal *expr* or it generates $\epsilon$. While parsing *optexpr*, if the lookahead symbol is not in FIRST(*expr*), then the $\epsilon$-production is used.

For more on when to use $\epsilon$-productions, see the discussion of LL(1) grammars in Section 4.4.3.

### 2.4.4 Designing a Predictive Parser

We can generalize the technique introduced informally in Section 2.4.2, to apply to any grammar that has disjoint FIRST sets for the production bodies belonging to any nonterminal. We shall also see that when we have a translation scheme — that is, a grammar with embedded actions — it is possible to execute those actions as part of the procedures designed for the parser.

Recall that a *predictive parser* is a program consisting of a procedure for every nonterminal. The procedure for nonterminal $A$ does two things.

1. It decides which $A$-production to use by examining the lookahead symbol. The production with body $\alpha$ (where $\alpha$ is not $\epsilon$, the empty string) is used if the lookahead symbol is in FIRST($\alpha$). If there is a conflict between two nonempty bodies for any lookahead symbol, then we cannot use this parsing method on this grammar. In addition, the $\epsilon$-production for $A$, if it exists, is used if the lookahead symbol is not in the FIRST set for any other production body for $A$.

2. The procedure then mimics the body of the chosen production. That is, the symbols of the body are "executed" in turn, from the left. A nonterminal is "executed" by a call to the procedure for that nonterminal, and a terminal matching the lookahead symbol is "executed" by reading the next input symbol. If at some point the terminal in the body does not match the lookahead symbol, a syntax error is reported.

Figure 2.19 is the result of applying these rules to the grammar in Fig. 2.16.

Just as a translation scheme is formed by extending a grammar, a syntax-directed translator can be formed by extending a predictive parser. An algorithm for this purpose is given in Section 5.4. The following limited construction suffices for the present:

1. Construct a predictive parser, ignoring the actions in productions.
2. Copy the actions from the translation scheme into the parser. If an action appears after grammar symbol $X$ in production $p$, then it is copied after the implementation of $X$ in the code for $p$. Otherwise, if it appears at the beginning of the production, then it is copied just before the code for the production body.

We shall construct such a translator in Section 2.5.

### 2.4.5 Left Recursion

It is possible for a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like

$$expr \quad \rightarrow \quad expr + term$$

where the leftmost symbol of the body is the same as the nonterminal at the head of the production. Suppose the procedure for *expr* decides to apply this production. The body begins with *expr* so the procedure for *expr* is called recursively. Since the lookahead symbol changes only when a terminal in the body is matched, no change to the input took place between recursive calls of *expr*. As a result, the second call to *expr* does exactly what the first call did, which means a third call to *expr*, and so on, forever.

A left-recursive production can be eliminated by rewriting the offending production. Consider a nonterminal $A$ with two productions

$$A \rightarrow A\alpha \mid \beta$$

where $\alpha$ and $\beta$ are sequences of terminals and nonterminals that do not start with $A$. For example, in

$$expr \quad \rightarrow \quad expr + term \quad \mid \quad term$$

nonterminal $A = expr$, string $\alpha = + term$, and string $\beta = term$.

The nonterminal $A$ and its production are said to be *left recursive*, because the production $A \rightarrow A\alpha$ has $A$ itself as the leftmost symbol on the right side.[4] Repeated application of this production builds up a sequence of $\alpha$'s to the right of $A$, as in Fig. 2.20(a). When $A$ is finally replaced by $\beta$, we have a $\beta$ followed by a sequence of zero or more $\alpha$'s.

The same effect can be achieved, as in Fig. 2.20(b), by rewriting the productions for $A$ in the following manner, using a new nonterminal $R$:

[4] In a general left-recursive grammar, instead of a production $A \rightarrow A\alpha$, the nonterminal $A$ may derive $A\alpha$ through intermediate productions.

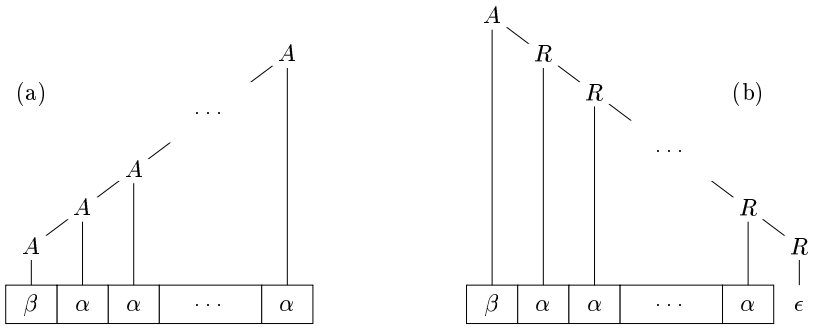



Figure 2.20: Left- and right-recursive ways of generating a string

$$\begin{array}{lcl} A & \rightarrow & \beta R \\ R & \rightarrow & \alpha R \mid \epsilon \end{array}$$

Nonterminal $R$ and its production $R \rightarrow \alpha R$ are *right recursive* because this production for $R$ has $R$ itself as the last symbol on the right side. Right-recursive productions lead to trees that grow down towards the right, as in Fig. 2.20(b). Trees growing down to the right make it harder to translate expressions containing left-associative operators, such as minus. In Section 2.5.2, however, we shall see that the proper translation of expressions into postfix notation can still be attained by a careful design of the translation scheme.

In Section 4.3.3, we shall consider more general forms of left recursion and show how all left recursion can be eliminated from a grammar.

### 2.4.6 Exercises for Section 2.4

**Exercise 2.4.1:** Construct recursive-descent parsers, starting with the following grammars:

a) $S \rightarrow$ **+** $S\ S \mid$ **-** $S\ S \mid$ **a**

b) $S \rightarrow S$ **(** $S$ **)** $S \mid \epsilon$

c) $S \rightarrow$ **0** $S$ **1** $\mid$ **0 1**

## 2.5 A Translator for Simple Expressions

Using the techniques of the last three sections, we now construct a syntax-directed translator, in the form of a working Java program, that translates arithmetic expressions into postfix form. To keep the initial program manageably small, we start with expressions consisting of digits separated by binary plus and minus signs. We extend the program in Section 2.6 to translate expressions that include numbers and other operators. It is worth studying the

translation of expressions in detail, since they appear as a construct in so many languages.

A syntax-directed translation scheme often serves as the specification for a translator. The scheme in Fig. 2.21 (repeated from Fig. 2.15) defines the translation to be performed here.

| | | | |
|---|---|---|---|
| *expr* | → | *expr* + *term* | { print('+') } |
| | \| | *expr* - *term* | { print('-') } |
| | \| | *term* | |
| *term* | → | 0 | { print('0') } |
| | \| | 1 | { print('1') } |
| | | ... | |
| | \| | 9 | { print('9') } |

Figure 2.21: Actions for translating into postfix notation

Often, the underlying grammar of a given scheme has to be modified before it can be parsed with a predictive parser. In particular, the grammar underlying the scheme in Fig. 2.21 is left recursive, and as we saw in the last section, a predictive parser cannot handle a left-recursive grammar.

We appear to have a conflict: on the one hand we need a grammar that facilitates translation, on the other hand we need a significantly different grammar that facilitates parsing. The solution is to begin with the grammar for easy translation and carefully transform it to facilitate parsing. By eliminating the left recursion in Fig. 2.21, we can obtain a grammar suitable for use in a predictive recursive-descent translator.

### 2.5.1 Abstract and Concrete Syntax

A useful starting point for designing a translator is a data structure called an abstract syntax tree. In an *abstract syntax tree* for an expression, each interior node represents an operator; the children of the node represent the operands of the operator. More generally, any programming construct can be handled by making up an operator for the construct and treating as operands the semantically meaningful components of that construct.

In the abstract syntax tree for `9-5+2` in Fig. 2.22, the root represents the operator `+`. The subtrees of the root represent the subexpressions `9-5` and `2`. The grouping of `9-5` as an operand reflects the left-to-right evaluation of operators at the same precedence level. Since `-` and `+` have the same precedence, `9-5+2` is equivalent to `(9-5)+2`.

Abstract syntax trees, or simply *syntax trees*, resemble parse trees to an extent. However, in the syntax tree, interior nodes represent programming constructs while in the parse tree, the interior nodes represent nonterminals. Many nonterminals of a grammar represent programming constructs, but others

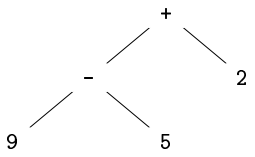



Figure 2.22: Syntax tree for `9-5+2`

are "helpers" of one sort of another, such as those representing terms, factors, or other variations of expressions. In the syntax tree, these helpers typically are not needed and are hence dropped. To emphasize the contrast, a parse tree is sometimes called a *concrete syntax tree*, and the underlying grammar is called a *concrete syntax* for the language.

In the syntax tree in Fig. 2.22, each interior node is associated with an operator, with no "helper" nodes for *single productions* (a production whose body consists of a single nonterminal, and nothing else) like $expr \rightarrow term$ or for $\epsilon$-productions like $rest \rightarrow \epsilon$.

It is desirable for a translation scheme to be based on a grammar whose parse trees are as close to syntax trees as possible. The grouping of subexpressions by the grammar in Fig. 2.21 is similar to their grouping in syntax trees. For example, subexpressions of the addition operator are given by *expr* and *term* in the production body *expr*+ *term*.

### 2.5.2 Adapting the Translation Scheme

The left-recursion-elimination technique sketched in Fig. 2.20 can also be applied to productions containing semantic actions. First, the technique extends to multiple productions for $A$. In our example, $A$ is *expr*, and there are two left-recursive productions for *expr* and one that is not left recursive. The technique transforms the productions $A \rightarrow A\alpha \mid A\beta \mid \gamma$ into

$$\begin{array}{lcl} A & \rightarrow & \gamma R \\ R & \rightarrow & \alpha R \mid \beta R \mid \epsilon \end{array}$$

Second, we need to transform productions that have embedded actions, not just terminals and nonterminals. Semantic actions embedded in the productions are simply carried along in the transformation, as if they were terminals.

**Example 2.13 :** Consider the translation scheme of Fig. 2.21. Let

$$\begin{array}{lcl} A & = & expr \\ \alpha & = & \texttt{+}\ term\ \{\ \text{print}(\texttt{'+'})\ \} \\ \beta & = & \texttt{-}\ term\ \{\ \text{print}(\texttt{'-'})\ \} \\ \gamma & = & term \end{array}$$

Then the left-recursion-eliminating transformation produces the translation scheme in Fig. 2.23. The *expr* productions in Fig. 2.21 have been transformed into one production for *expr*, and a new nonterminal *rest* plays the role of $R$. The productions for *term* are repeated from Fig. 2.21. Figure 2.24 shows how `9-5+2` is translated using the grammar in Fig. 2.23. □

$$
\begin{array}{lcl}
\mathit{expr} & \rightarrow & \mathit{term}\ \mathit{rest} \\
\mathit{rest} & \rightarrow & \texttt{+}\ \mathit{term}\ \{\ \mathrm{print}(\texttt{'+'})\ \}\ \mathit{rest} \\
 & | & \texttt{-}\ \mathit{term}\ \{\ \mathrm{print}(\texttt{'-'})\ \}\ \mathit{rest} \\
 & | & \epsilon \\
\mathit{term} & \rightarrow & \texttt{0}\ \{\ \mathrm{print}(\texttt{'0'})\ \} \\
 & | & \texttt{1}\ \{\ \mathrm{print}(\texttt{'1'})\ \} \\
 & & \cdots \\
 & | & \texttt{9}\ \{\ \mathrm{print}(\texttt{'9'})\ \}
\end{array}
$$

Figure 2.23: Translation scheme after left-recursion elimination

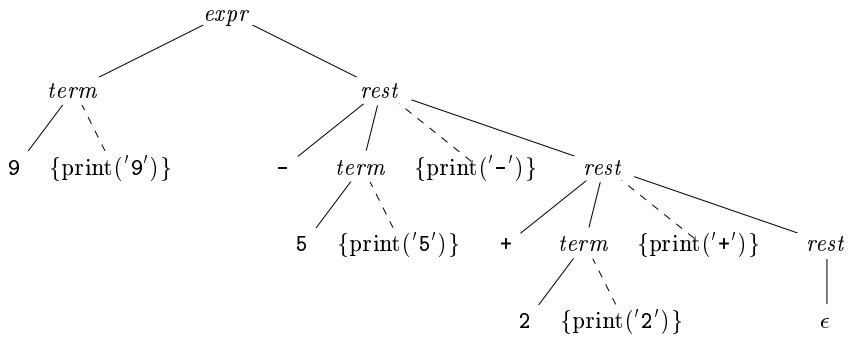



Figure 2.24: Translation of `9-5+2` to `95-2+`

Left-recursion elimination must be done carefully, to ensure that we preserve the ordering of semantic actions. For example, the transformed scheme in Fig. 2.23 has the actions { print('+') } and { print('-') } in the middle of a production body, in each case between nonterminals *term* and *rest*. If the actions were to be moved to the end, after *rest*, then the translations would become incorrect. We leave it to the reader to show that `9-5+2` would then be translated incorrectly into `952+-`, the postfix notation for `9-(5+2)`, instead of the desired `95-2+`, the postfix notation for `(9-5)+2`.

### 2.5.3 Procedures for the Nonterminals

Functions *expr*, *rest*, and *term* in Fig. 2.25 implement the syntax-directed translation scheme in Fig. 2.23. These functions mimic the production bodies of the corresponding nonterminals. Function *expr* implements the production $expr \rightarrow term\ rest$ by the calls *term*() followed by *rest*().

```
void expr() {
        term(); rest();
}

void rest() {
        if ( lookahead == '+' ) {
                match('+'); term(); print('+'); rest();
        }
        else if ( lookahead == '-' ) {
                match('-'); term(); print('-'); rest();
        }
        else { } /* do nothing with the input */ ;
}

void term() {
        if ( lookahead is a digit ) {
                t = lookahead; match(lookahead); print(t);
        }
        else report("syntax error");
}
```

Figure 2.25: Pseudocode for nonterminals *expr*, *rest*, and *term*.

Function *rest* implements the three productions for nonterminal *rest* in Fig. 2.23. It applies the first production if the lookahead symbol is a plus sign, the second production if the lookahead symbol is a minus sign, and the production $rest \rightarrow \epsilon$ in all other cases. The first two productions for *rest* are implemented by the first two branches of the if-statement in procedure *rest*. If the lookahead symbol is +, the plus sign is matched by the call *match*('+'). After the call *term*(), the semantic action is implemented by writing a plus character. The second production is similar, with - instead of +. Since the third production for *rest* has $\epsilon$ as its right side, the last else-clause in function *rest* does nothing.

The ten productions for *term* generate the ten digits. Since each of these productions generates a digit and prints it, the same code in Fig. 2.25 implements them all. If the test succeeds, variable $t$ saves the digit represented by *lookahead* so it can be written after the call to *match*. Note that *match* changes

the lookahead symbol, so the digit needs to be saved for later printing.[5]

### 2.5.4 Simplifying the Translator

Before showing a complete program, we shall make two simplifying transformations to the code in Fig. 2.25. The simplifications will fold procedure *rest* into procedure *expr*. When expressions with multiple levels of precedence are translated, such simplifications reduce the number of procedures needed.

First, certain recursive calls can be replaced by iterations. When the last statement executed in a procedure body is a recursive call to the same procedure, the call is said to be *tail recursive*. For example, in function *rest*, the calls of *rest*() with lookahead + and - are tail recursive because in each of these branches, the recursive call to *rest* is the last statement executed by the given call of *rest*.

For a procedure without parameters, a tail-recursive call can be replaced simply by a jump to the beginning of the procedure. The code for *rest* can be rewritten as the pseudocode of Fig. 2.26. As long as the lookahead symbol is a plus or a minus sign, procedure *rest* matches the sign, calls *term* to match a digit, and continues the process. Otherwise, it breaks out of while loop and returns from *rest*.

```
void rest() {
    while( true ) {
        if( lookahead == '+' ) {
            match('+'); term(); print('+'); continue;
        }
        else if ( lookahead == '-' ) {
            match('-'); term(); print('-'); continue;
        }
        break ;
    }
}
```

Figure 2.26: Eliminating tail recursion in the procedure *rest* of Fig. 2.25.

Second, the complete Java program will include one more change. Once the tail-recursive calls to *rest* in Fig. 2.25 are replaced by iterations, the only remaining call to *rest* is from within procedure *expr*. The two procedures can therefore be integrated into one, by replacing the call *rest*() by the body of procedure *rest*.

[5] As a minor optimization, we could print before calling *match* to avoid the need to save the digit. In general, changing the order of actions and grammar symbols is risky, since it could change what the translation does.

### 2.5.5 The Complete Program

The complete Java program for our translator appears in Fig. 2.27. The first line of Fig. 2.27, beginning with `import`, provides access to the package `java.io` for system input and output. The rest of the code consists of the two classes `Parser` and `Postfix`. Class `Parser` contains variable `lookahead` and functions `Parser`, `expr`, `term`, and `match`.

Execution begins with function `main`, which is defined in class `Postfix`. Function `main` creates an instance `parse` of class `Parser` and calls its function `expr` to parse an expression.

The function `Parser`, with the same name as its class, is a *constructor*; it is called automatically when an object of the class is created. Notice from its definition at the beginning of class `Parser` that the constructor `Parser` initializes variable `lookahead` by reading a token. Tokens, consisting of single characters, are supplied by the system input routine `read`, which reads the next character from the input file. Note that `lookahead` is declared to be an integer, rather than a character, to anticipate the fact that additional tokens other than single characters will be introduced in later sections.

Function `expr` is the result of the simplifications discussed in Section 2.5.4; it implements nonterminals *expr* and *rest* in Fig. 2.23. The code for `expr` in Fig. 2.27 calls `term` and then has a while-loop that forever tests whether `lookahead` matches either `'+'` or `'-'`. Control exits from this while-loop when it reaches the return statement. Within the loop, the input/output facilities of the `System` class are used to write a character.

Function `term` uses the routine `isDigit` from the Java class `Character` to test if the lookahead symbol is a digit. The routine `isDigit` expects to be applied to a character; however, `lookahead` is declared to be an integer, anticipating future extensions. The construction `(char)lookahead` *casts* or coerces `lookahead` to be a character. In a small change from Fig. 2.25, the semantic action of writing the lookahead character occurs before the call to `match`.

The function `match` checks terminals; it reads the next input terminal if the lookahead symbol is matched and signals an error otherwise by executing

```
throw new Error("syntax error");
```

This code creates a new exception of class `Error` and supplies it the string `syntax error` as an error message. Java does not require `Error` exceptions to be declared in a `throws` clause, since they are meant to be used only for abnormal events that should never occur.[6]

[6] Error handling can be streamlined using the exception-handling facilities of Java. One approach is to define a new exception, say `SyntaxError`, that extends the system class `Exception`. Then, throw `SyntaxError` instead of `Error` when an error is detected in either `term` or `match`. Further, handle the exception in `main` by enclosing the call `parse.expr()` within a `try` statement that catches exception `SyntaxError`, writes a message, and terminates. We would need to add a class `SyntaxError` to the program in Fig. 2.27. To complete the extension, in addition to `IOException`, functions `match` and `term` must now declare that they can throw `SyntaxError`. Function `expr`, which calls them, must also declare that it can throw `SyntaxError`.

```
import java.io.*;
class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void expr() throws IOException {
        term();
        while(true) {
            if( lookahead == '+' ) {
                match('+'); term(); System.out.write('+');
            }
            else if( lookahead == '-' ) {
                match('-'); term(); System.out.write('-');
            }
            else return;
        }
    }

    void term() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.write((char)lookahead); match(lookahead);
        }
        else throw new Error("syntax error");
    }

    void match(int t) throws IOException {
        if( lookahead == t ) lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}

public class Postfix {
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.expr(); System.out.write('\n');
    }
}
```

Figure 2.27: Java program to translate infix expressions into postfix form

### A Few Salient Features of Java

Those unfamiliar with Java may find the following notes on Java helpful in reading the code in Fig. 2.27:

- A class in Java consists of a sequence of variable and function definitions.
- Parentheses enclosing function parameter lists are needed even if there are no parameters; hence we write `expr()` and `term()`. These functions are actually procedures, because they do not return values, signified by the keyword `void` before the function name.
- Functions communicate either by passing parameters "by value" or by accessing shared data. For example, the functions `expr()` and `term()` examine the lookahead symbol using the class variable `lookahead` that they can all access since they all belong to the same class `Parser`.
- Like C, Java uses `=` for assignment, `==` for equality, and `!=` for inequality.
- The clause "`throws IOException`" in the definition of `term()` declares that an exception called `IOException` can occur. Such an exception occurs if there is no input to be read when the function `match` uses the routine `read`. Any function that calls `match` must also declare that an `IOException` can occur during its own execution.

## 2.6 Lexical Analysis

A lexical analyzer reads characters from the input and groups them into "token objects." Along with a terminal symbol that is used for parsing decisions, a token object carries additional information in the form of attribute values. So far, there has been no need to distinguish between the terms "token" and "terminal," since the parser ignores the attribute values that are carried by a token. In this section, a token is a terminal along with additional information.

A sequence of input characters that comprises a single token is called a *lexeme*. Thus, we can say that the lexical analyzer insulates a parser from the lexeme representation of tokens.

The lexical analyzer in this section allows numbers, identifiers, and "white space" (blanks, tabs, and newlines) to appear within expressions. It can be used to extend the expression translator of the previous section. Since the expression grammar of Fig. 2.21 must be extended to allow numbers and identifiers, we

shall take this opportunity to allow multiplication and division as well. The extended translation scheme appears in Fig. 2.28.

$$
\begin{array}{lcll}
expr & \rightarrow & expr + term & \{ \text{print}('+') \} \\
 & | & expr - term & \{ \text{print}('-') \} \\
 & | & term & \\
 & & & \\
term & \rightarrow & term * factor & \{ \text{print}('*') \} \\
 & | & term / factor & \{ \text{print}('/') \} \\
 & | & factor & \\
 & & & \\
factor & \rightarrow & ( \; expr \; ) & \\
 & | & \textbf{num} & \{ \text{print}(\textbf{num}.value) \} \\
 & | & \textbf{id} & \{ \text{print}(\textbf{id}.lexeme) \}
\end{array}
$$

Figure 2.28: Actions for translating into postfix notation

In Fig. 2.28, the terminal **num** is assumed to have an attribute **num**.*value*, which gives the integer value corresponding to this occurrence of **num**. Terminal **id** has a string-valued attribute written as **id**.*lexeme*; we assume this string is the actual lexeme comprising this instance of the token **id**.

The pseudocode fragments used to illustrate the workings of a lexical analyzer will be assembled into Java code at the end of this section. The approach in this section is suitable for hand-written lexical analyzers. Section 3.5 describes a tool called Lex that generates a lexical analyzer from a specification. Symbol tables or data structures for holding information about identifiers are considered in Section 2.7.

### 2.6.1 Removal of White Space and Comments

The expression translator in Section 2.5 sees every character in the input, so extraneous characters, such as blanks, will cause it to fail. Most languages allow arbitrary amounts of white space to appear between tokens. Comments are likewise ignored during parsing, so they may also be treated as white space.

If white space is eliminated by the lexical analyzer, the parser will never have to consider it. The alternative of modifying the grammar to incorporate white space into the syntax is not nearly as easy to implement.

The pseudocode in Fig. 2.29 skips white space by reading input characters as long as it sees a blank, a tab, or a newline. Variable *peek* holds the next input character. Line numbers and context are useful within error messages to help pinpoint errors; the code uses variable *line* to count newline characters in the input.

```
for ( ; ; peek = next input character ) {
        if ( peek is a blank or a tab ) do nothing;
        else if ( peek is a newline ) line = line+1;
        else break;
}
```

Figure 2.29: Skipping white space

### 2.6.2 Reading Ahead

A lexical analyzer may need to read ahead some characters before it can decide on the token to be returned to the parser. For example, a lexical analyzer for C or Java must read ahead after it sees the character `>`. If the next character is `=`, then `>` is part of the character sequence `>=`, the lexeme for the token for the "greater than or equal to" operator. Otherwise `>` itself forms the "greater than" operator, and the lexical analyzer has read one character too many.

A general approach to reading ahead on the input, is to maintain an input buffer from which the lexical analyzer can read and push back characters. Input buffers can be justified on efficiency grounds alone, since fetching a block of characters is usually more efficient than fetching one character at a time. A pointer keeps track of the portion of the input that has been analyzed; pushing back a character is implemented by moving back the pointer. Techniques for input buffering are discussed in Section 3.2.

One-character read-ahead usually suffices, so a simple solution is to use a variable, say *peek*, to hold the next input character. The lexical analyzer in this section reads ahead one character while it collects digits for numbers or characters for identifiers; e.g., it reads past `1` to distinguish between `1` and `10`, and it reads past `t` to distinguish between `t` and `true`.

The lexical analyzer reads ahead only when it must. An operator like `*` can be identified without reading ahead. In such cases, *peek* is set to a blank, which will be skipped when the lexical analyzer is called to find the next token. The invariant assertion in this section is that when the lexical analyzer returns a token, variable *peek* either holds the character beyond the lexeme for the current token, or it holds a blank.

### 2.6.3 Constants

Anytime a single digit appears in a grammar for expressions, it seems reasonable to allow an arbitrary integer constant in its place. Integer constants can be allowed either by creating a terminal symbol, say **num**, for such constants or by incorporating the syntax of integer constants into the grammar. The job of collecting characters into integers and computing their collective numerical value is generally given to a lexical analyzer, so numbers can be treated as single units during parsing and translation.

When a sequence of digits appears in the input stream, the lexical analyzer passes to the parser a token consisting of the terminal **num** along with an integer-valued attribute computed from the digits. If we write tokens as tuples enclosed between $\langle\ \rangle$, the input `31 + 28 + 59` is transformed into the sequence

$$\langle \mathbf{num}, 31\rangle\ \ \langle \texttt{+}\rangle\ \ \langle \mathbf{num}, 28\rangle\ \ \langle \texttt{+}\rangle\ \ \langle \mathbf{num}, 59\rangle$$

Here, the terminal symbol `+` has no attributes, so its tuple is simply $\langle$`+`$\rangle$. The pseudocode in Fig. 2.30 reads the digits in an integer and accumulates the value of the integer using variable $v$.

```
if ( peek holds a digit ) {
        v = 0;
        do {
                v = v * 10 + integer value of digit peek;
                peek = next input character;
        } while ( peek holds a digit );
        return token ⟨num, v⟩;
}
```

Figure 2.30: Grouping digits into integers

### 2.6.4 Recognizing Keywords and Identifiers

Most languages use fixed character strings such as `for`, `do`, and `if`, as punctuation marks or to identify constructs. Such character strings are called *keywords*.

Character strings are also used as identifiers to name variables, arrays, functions, and the like. Grammars routinely treat identifiers as terminals to simplify the parser, which can then expect the same terminal, say **id**, each time any identifier appears in the input. For example, on input

```
count = count + increment;
```
(2.6)

the parser works with the terminal stream **id** `=` **id** `+` **id**. The token for **id** has an attribute that holds the lexeme. Writing tokens as tuples, we see that the tuples for the input stream (2.6) are

$$\langle \mathbf{id}, \texttt{"count"}\rangle\ \langle \texttt{=}\rangle\ \langle \mathbf{id}, \texttt{"count"}\rangle\ \langle \texttt{+}\rangle\ \langle \mathbf{id}, \texttt{"increment"}\rangle\ \langle \texttt{;}\rangle$$

.

Keywords generally satisfy the rules for forming identifiers, so a mechanism is needed for deciding when a lexeme forms a keyword and when it forms an identifier. The problem is easier to resolve if keywords are *reserved*; i.e., if they cannot be used as identifiers. Then, a character string forms an identifier only if it is not a keyword.

The lexical analyzer in this section solves two problems by using a table to hold character strings:

- *Single Representation.* A string table can insulate the rest of the compiler from the representation of strings, since the phases of the compiler can work with references or pointers to the string in the table. References can also be manipulated more efficiently than the strings themselves.

- *Reserved Words.* Reserved words can be implemented by initializing the string table with the reserved strings and their tokens. When the lexical analyzer reads a string or lexeme that could form an identifier, it first checks whether the lexeme is in the string table. If so, it returns the token from the table; otherwise, it returns a token with terminal **id**.

In Java, a string table can be implemented as a hash table using a class called *Hashtable*. The declaration

$$Hashtable\ words = \textbf{new}\ Hashtable();$$

sets up *words* as a default hash table that maps keys to values. We shall use it to map lexemes to tokens. The pseudocode in Fig. 2.31 uses the operation *get* to look up reserved words.

```
if ( peek holds a letter ) {
        collect letters or digits into a buffer b;
        s = string formed from the characters in b;
        w = token returned by words.get(s);
        if ( w is not null ) return w;
        else {
                Enter the key-value pair (s, ⟨id, s⟩) into words
                return token ⟨id, s⟩;
        }
}
```

Figure 2.31: Distinguishing keywords from identifiers

This pseudocode collects from the input a string $s$ consisting of letters and digits beginning with a letter. We assume that $s$ is made as long as possible; i.e., the lexical analyzer will continue reading from the input as long as it encounters letters and digits. When something other than a letter or digit, e.g., white space, is encountered, the lexeme is copied into a buffer $b$. If the table has an entry for $s$, then the token retrieved by *words.get* is returned. Here, $s$ could be either a keyword, with which the *words* table was initially seeded, or it could be an identifier that was previously entered into the table. Otherwise, token **id** and attribute $s$ are installed in the table and returned.

### 2.6.5 A Lexical Analyzer

The pseudocode fragments so far in this section fit together to form a function *scan* that returns token objects, as follows:

```
Token scan() {
    skip white space, as in Section 2.6.1;
    handle numbers, as in Section 2.6.3;
    handle reserved words and identifiers, as in Section 2.6.4;
    /* if we get here, treat read-ahead character peek as a token */
    Token t = new Token(peek);
    peek = blank /* initialization, as discussed in Section 2.6.2 */ ;
    return t;
}
```

The rest of this section implements function *scan* as part of a Java package for lexical analysis. The package, called `lexer` has classes for tokens and a class `Lexer` containing function `scan`.

The classes for tokens and their fields are illustrated in Fig. 2.32; their methods are not shown. Class `Token` has a field `tag` that is used for parsing decisions. Subclass `Num` adds a field `value` for an integer value. Subclass `Word` adds a field `lexeme` that is used for reserved words and identifiers.

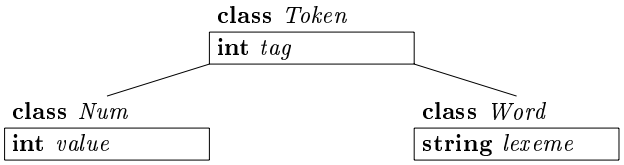



Figure 2.32: Class *Token* and subclasses *Num* and *Word*

Each class is in a file by itself. The file for class `Token` is as follows:

```
1) package lexer;                    // File Token.java
2) public class Token {
3)    public final int tag;
4)    public Token(int t) { tag = t; }
5) }
```

Line 1 identifies the package `lexer`. Field `tag` is declared on line 3 to be `final` so it cannot be changed once it is set. The constructor `Token` on line 4 is used to create token objects, as in

```
new Token('+')
```

which creates a new object of class `Token` and sets its field `tag` to an integer representation of `'+'`. (For brevity, we omit the customary method `toString`, which would return a string suitable for printing.)

Where the pseudocode had terminals like **num** and **id**, the Java code uses integer constants. Class `Tag` implements such constants:

```
1) package lexer;                    // File Tag.java
2) public class Tag {
3)    public final static int
4)       NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
5) }
```

In addition to the integer-valued fields `NUM` and `ID`, this class defines two additional fields, `TRUE` and `FALSE`, for future use; they will be used to illustrate the treatment of reserved keywords.[7]

The fields in class `Tag` are `public`, so they can be used outside the package. They are `static`, so there is just one instance or copy of these fields. The fields are `final`, so they can be set just once. In effect, these fields represent constants. A similar effect is achieved in C by using define-statements to allow names such as `NUM` to be used as symbolic constants, e.g.:

```
#define NUM 256
```

The Java code refers to `Tag.NUM` and `Tag.ID` in places where the pseudocode referred to terminals **num** and **id**. The only requirement is that `Tag.NUM` and `Tag.ID` must be initialized with distinct values that differ from each other and from the constants representing single-character tokens, such as `'+'` or `'*'`.

```
1) package lexer;                    // File Num.java
2) public class Num extends Token {
3)    public final int value;
4)    public Num(int v) { super(Tag.NUM); value = v; }
5) }
```

```
1) package lexer;                    // File Word.java
2) public class Word extends Token {
3)    public final String lexeme;
4)    public Word(int t, String s) {
5)       super(t); lexeme = new String(s);
6)    }
7) }
```

Figure 2.33: Subclasses `Num` and `Word` of `Token`

Classes `Num` and `Word` appear in Fig. 2.33. Class `Num` extends `Token` by declaring an integer field `value` on line 3. The constructor `Num` on line 4 calls `super(Tag.NUM)`, which sets field `tag` in the superclass `Token` to `Tag.NUM`.

[7] ASCII characters are typically converted into integers between 0 and 255. We therefore use integers greater than 255 for terminals.

```
 1) package lexer;                    // File Lexer.java
 2) import java.io.*; import java.util.*;
 3) public class Lexer {
 4)    public int line = 1;
 5)    private char peek = ' ';
 6)    private Hashtable words = new Hashtable();
 7)    void reserve(Word t) { words.put(t.lexeme, t); }
 8)    public Lexer() {
 9)       reserve( new Word(Tag.TRUE,  "true")  );
10)       reserve( new Word(Tag.FALSE, "false") );
11)    }
12)    public Token scan() throws IOException {
13)       for( ; ; peek = (char)System.in.read() ) {
14)          if( peek == ' ' || peek == '\t' ) continue;
15)          else if( peek == '\n' ) line = line + 1;
16)          else break;
17)       }
          /* continues in Fig. 2.35 */
```

Figure 2.34: Code for a lexical analyzer, part 1 of 2

Class `Word` is used for both reserved words and identifiers, so the constructor `Word` on line 4 expects two parameters: a lexeme and a corresponding integer value for `tag`. An object for the reserved word `true` can be created by executing

```
new Word(Tag.TRUE, "true")
```

which creates a new object with field `tag` set to `Tag.TRUE` and field `lexeme` set to the string `"true"`.

Class `Lexer` for lexical analysis appears in Figs. 2.34 and 2.35. The integer variable `line` on line 4 counts input lines, and character variable `peek` on line 5 holds the next input character.

Reserved words are handled on lines 6 through 11. The table `words` is declared on line 6. The helper function `reserve` on line 7 puts a string-word pair in the table. Lines 9 and 10 in the constructor `Lexer` initialize the table. They use the constructor `Word` to create word objects, which are passed to the helper function `reserve`. The table is therefore initialized with reserved words `"true"` and `"false"` before the first call of `scan`.

The code for `scan` in Fig. 2.34–2.35 implements the pseudocode fragments in this section. The for-statement on lines 13 through 17 skips blank, tab, and newline characters. Control leaves the for-statement with `peek` holding a non-white-space character.

The code for reading a sequence of digits is on lines 18 through 25. The function `isDigit` is from the built-in Java class `Character`. It is used on line 18 to check whether `peek` is a digit. If so, the code on lines 19 through 24

```
18)        if( Character.isDigit(peek) ) {
19)           int v = 0;
20)           do {
21)              v = 10*v + Character.digit(peek, 10);
22)              peek = (char)System.in.read();
23)           } while( Character.isDigit(peek) );
24)           return new Num(v);
25)        }
26)        if( Character.isLetter(peek) ) {
27)           StringBuffer b = new StringBuffer();
28)           do {
29)              b.append(peek);
30)              peek = (char)System.in.read();
31)           } while( Character.isLetterOrDigit(peek) );
32)           String s = b.toString();
33)           Word w = (Word)words.get(s);
34)           if( w != null ) return w;
35)           w = new Word(Tag.ID, s);
36)           words.put(s, w);
37)           return w;
38)        }
39)        Token t = new Token(peek);
40)        peek = ' ';
41)        return t;
42)     }
43)  }
```

Figure 2.35: Code for a lexical analyzer, part 2 of 2

accumulates the integer value of the sequence of digits in the input and returns a new `Num` object.

Lines 26 through 38 analyze reserved words and identifiers. Keywords **true** and **false** have already been reserved on lines 9 and 10. Therefore, line 35 is reached if string `s` is not reserved, so it must be the lexeme for an identifier. Line 35 therefore returns a new word object with `lexeme` set to `s` and `tag` set to `Tag.ID`. Finally, lines 39 through 41 return the current character as a token and set `peek` to a blank that will be stripped the next time `scan` is called.

### 2.6.6 Exercises for Section 2.6

**Exercise 2.6.1:** Extend the lexical analyzer in Section 2.6.5 to remove comments, defined as follows:

a) A comment begins with `//` and includes all characters until the end of that line.

b) A comment begins with `/*` and includes all characters through the next occurrence of the character sequence `*/`.

**Exercise 2.6.2:** Extend the lexical analyzer in Section 2.6.5 to recognize the relational operators `<`, `<=`, `==`, `!=`, `>=`, `>`.

**Exercise 2.6.3:** Extend the lexical analyzer in Section 2.6.5 to recognize floating point numbers such as `2.`, `3.14`, and `.5`.

## 2.7 Symbol Tables

*Symbol tables* are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.

From Section 1.6.1, the scope of a declaration is the portion of a program to which the declaration applies. We shall implement scopes by setting up a separate symbol table for each scope. A program block with declarations[8] will have its own symbol table with an entry for each declaration in the block. This approach also works for other constructs that set up scopes; for example, a class would have its own table, with an entry for each field and method.

This section contains a symbol-table module suitable for use with the Java translator fragments in this chapter. The module will be used as is when we put together the translator in Appendix A. Meanwhile, for simplicity, the main example of this section is a stripped-down language with just the key constructs that touch symbol tables; namely, blocks, declarations, and factors. All of the other statement and expression constructs are omitted so we can focus on the symbol-table operations. A program consists of blocks with optional declarations and "statements" consisting of single identifiers. Each such statement represents a use of the identifier. Here is a sample program in this language:

```
{ int x; char y; { bool y; x; y; } x; y; }        (2.7)
```

The examples of block structure in Section 1.6.3 dealt with the definitions and uses of names; the input (2.7) consists solely of definitions and uses of names.

The task we shall perform is to print a revised program, in which the declarations have been removed and each "statement" has its identifier followed by a colon and its type.

[8] In C, for instance, program blocks are either functions or sections of functions that are separated by curly braces and that have one or more declarations within them.

### Who Creates Symbol-Table Entries?

Symbol-table entries are created and used during the analysis phase by the lexical analyzer, the parser, and the semantic analyzer. In this chapter, we have the parser create entries. With its knowledge of the syntactic structure of a program, a parser is often in a better position than the lexical analyzer to distinguish among different declarations of an identifier.

In some cases, a lexical analyzer can create a symbol-table entry as soon as it sees the characters that make up a lexeme. More often, the lexical analyzer can only return to the parser a token, say **id**, along with a pointer to the lexeme. Only the parser, however, can decide whether to use a previously created symbol-table entry or create a new one for the identifier.

**Example 2.14:** On the above input (2.7), the goal is to produce:

```
{ { x:int; y:bool; } x:int; y:char; }
```

The first `x` and `y` are from the inner block of input (2.7). Since this use of `x` refers to the declaration of `x` in the outer block, it is followed by `int`, the type of that declaration. The use of `y` in the inner block refers to the declaration of `y` in that very block and therefore has boolean type. We also see the uses of `x` and `y` in the outer block, with their types, as given by declarations of the outer block: integer and character, respectively. □

### 2.7.1 Symbol Table Per Scope

The term "scope of identifier $x$" really refers to the scope of a particular declaration of $x$. The term *scope* by itself refers to a portion of a program that is the scope of one or more declarations.

Scopes are important, because the same identifier can be declared for different purposes in different parts of a program. Common names like `i` and `x` often have multiple uses. As another example, subclasses can redeclare a method name to override a method in a superclass.

If blocks can be nested, several declarations of the same identifier can appear within a single block. The following syntax results in nested blocks when *stmts* can generate a block:

$$block \rightarrow \text{'}\{\text{'}\ decls\ stmts\ \text{'}\}\text{'}$$

(We quote curly braces in the syntax to distinguish them from curly braces for semantic actions.) With the grammar in Fig. 2.38, *decls* generates an optional sequence of declarations and *stmts* generates an optional sequence of statements.

### Optimization of Symbol Tables for Blocks

Implementations of symbol tables for blocks can take advantage of the most-closely nested rule. Nesting ensures that the chain of applicable symbol tables forms a stack. At the top of the stack is the table for the current block. Below it in the stack are the tables for the enclosing blocks. Thus, symbol tables can be allocated and deallocated in a stack-like fashion.

Some compilers maintain a single hash table of accessible entries; that is, of entries that are not hidden by a declaration in a nested block. Such a hash table supports essentially constant-time lookups, at the expense of inserting and deleting entries on block entry and exit. Upon exit from a block $B$, the compiler must undo any changes to the hash table due to declarations in block $B$. It can do so by using an auxiliary stack to keep track of changes to the hash table while block $B$ is processed.

Moreover, a statement can be a block, so our language allows nested blocks, where an identifier can be redeclared.

The *most-closely nested* rule for blocks is that an identifier $x$ is in the scope of the most-closely nested declaration of $x$; that is, the declaration of $x$ found by examining blocks inside-out, starting with the block in which $x$ appears.

**Example 2.15:** The following pseudocode uses subscripts to distinguish among distinct declarations of the same identifier:

```
1)  {    int x1; int y1;
2)       {    int w2; bool y2; int z2;
3)            ··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
4)       }
5)       ··· w0 ···; ··· x1 ···; ··· y1 ···;
6)  }
```

The subscript is not part of an identifier; it is in fact the line number of the declaration that applies to the identifier. Thus, all occurrences of $x$ are within the scope of the declaration on line 1. The occurrence of $y$ on line 3 is in the scope of the declaration of $y$ on line 2 since $y$ is redeclared within the inner block. The occurrence of $y$ on line 5, however, is within the scope of the declaration of $y$ on line 1.

The occurrence of $w$ on line 5 is presumably within the scope of a declaration of $w$ outside this program fragment; its subscript 0 denotes a declaration that is global or external to this block.

Finally, $z$ is declared and used within the nested block, but cannot be used on line 5, since the nested declaration applies only to the nested block. □

The most-closely nested rule for blocks can be implemented by chaining symbol tables. That is, the table for a nested block points to the table for its enclosing block.

**Example 2.16 :** Figure 2.36 shows symbol tables for the pseudocode in Example 2.15. $B_1$ is for the block starting on line 1 and $B_2$ is for the block starting at line 2. At the top of the figure is an additional symbol table $B_0$ for any global or default declarations provided by the language. During the time that we are analyzing lines 2 through 4, the environment is represented by a reference to the lowest symbol table — the one for $B_2$. When we move to line 5, the symbol table for $B_2$ becomes inaccessible, and the environment refers instead to the symbol table for $B_1$, from which we can reach the global symbol table, but not the table for $B_2$. □

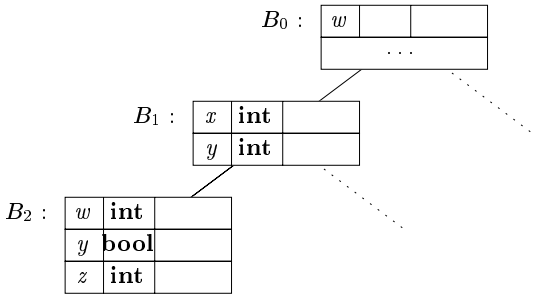



Figure 2.36: Chained symbol tables for Example 2.15

The Java implementation of chained symbol tables in Fig. 2.37 defines a class `Env`, short for *environment*.[9] Class `Env` supports three operations:

- *Create a new symbol table.* The constructor `Env(p)` on lines 6 through 8 of Fig. 2.37 creates an `Env` object with a hash table named `table`. The object is chained to the environment-valued parameter `p` by setting field `prev` to `p`. Although it is the `Env` objects that form a chain, it is convenient to talk of the tables being chained.
- *Put* a new entry in the current table. The hash table holds key-value pairs, where:
  - The *key* is a string, or rather a reference to a string. We could alternatively use references to token objects for identifiers as keys.
  - The *value* is an entry of class `Symbol`. The code on lines 9 through 11 does not need to know the structure of an entry; that is, the code is independent of the fields and methods in class `Symbol`.

[9] "Environment" is another term for the collection of symbol tables that are relevant at a point in the program.

```
 1) package symbols;                    // File Env.java
 2) import java.util.*;
 3) public class Env {
 4)    private Hashtable table;
 5)    protected Env prev;

 6)    public Env(Env p) {
 7)       table = new Hashtable(); prev = p;
 8)    }

 9)    public void put(String s, Symbol sym) {
10)       table.put(s, sym);
11)    }

12)    public Symbol get(String s) {
13)       for( Env e = this; e != null; e = e.prev ) {
14)          Symbol found = (Symbol)(e.table.get(s));
15)          if( found != null ) return found;
16)       }
17)       return null;
18)    }
19) }
```

Figure 2.37: Class *Env* implements chained symbol tables

- *Get* an entry for an identifier by searching the chain of tables, starting with the table for the current block. The code for this operation on lines 12 through 18 returns either a symbol-table entry or `null`.

Chaining of symbol tables results in a tree structure, since more than one block can be nested inside an enclosing block. The dotted lines in Fig. 2.36 are a reminder that chained symbol tables can form a tree.

### 2.7.2 The Use of Symbol Tables

In effect, the role of a symbol table is to pass information from declarations to uses. A semantic action "puts" information about identifier $x$ into the symbol table, when the declaration of $x$ is analyzed. Subsequently, a semantic action associated with a production such as $factor \rightarrow \mathbf{id}$ "gets" information about the identifier from the symbol table. Since the translation of an expression $E_1$ **op** $E_2$, for a typical operator **op**, depends only on the translations of $E_1$ and $E_2$, and does not directly depend on the symbol table, we can add any number of operators without changing the basic flow of information from declarations to uses, through the symbol table.

**Example 2.17:** The translation scheme in Fig. 2.38 illustrates how class *Env* can be used. The translation scheme concentrates on scopes, declarations, and

uses. It implements the translation described in Example 2.14. As noted earlier, on input

```
program → block                                  { top = null; }

block   → '{'                                    { saved = top;
                                                   top = new Env(top);
                                                   print("{ "); }
          decls stmts '}'                        { top = saved;
                                                   print("} "); }

decls   → decls decl
        | ϵ

decl    → type id ;                              { s = new Symbol;
                                                   s.type = type.lexeme
                                                   top.put(id.lexeme, s); }

stmts   → stmts stmt
        | ϵ

stmt    → block
        | factor ;                               { print("; "); }

factor  → id                                     { s = top.get(id.lexeme);
                                                   print(id.lexeme);
                                                   print(":"); }
                                                   print(s.type);
```

Figure 2.38: The use of symbol tables for translating a language with blocks

```
{ int x; char y; { bool y; x; y; } x; y; }
```

the translation scheme strips the declarations and produces

```
{ { x:int; y:bool; } x:int; y:char; }
```

Notice that the bodies of the productions have been aligned in Fig. 2.38 so that all the grammar symbols appear in one column, and all the actions in a second column. As a result, components of the body are often spread over several lines.

Now, consider the semantic actions. The translation scheme creates and discards symbol tables upon block entry and exit, respectively. Variable *top* denotes the top table, at the head of a chain of tables. The first production of