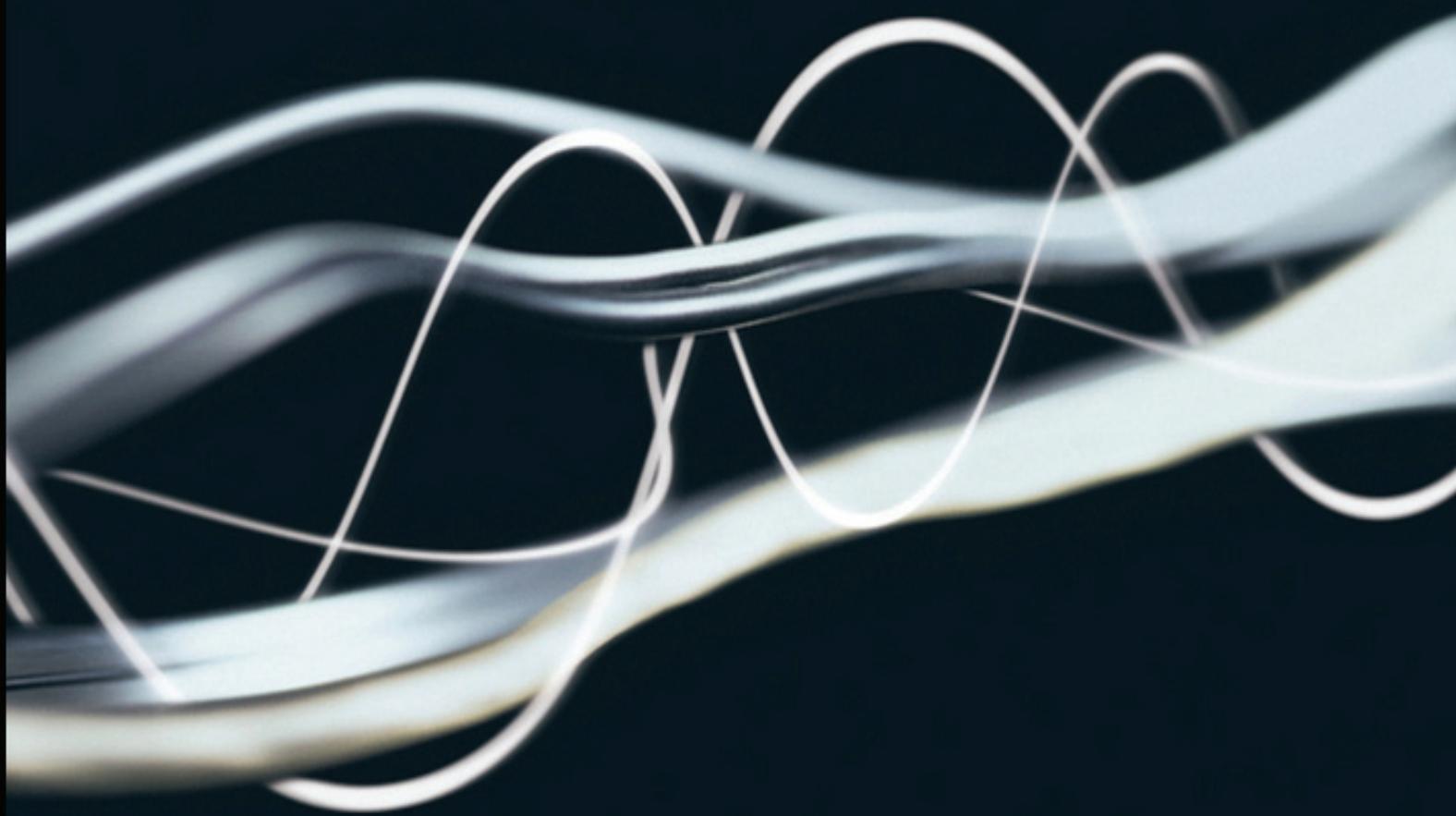# Writing
Compilers and Interpreters
A Software Engineering Approach

Ronald Mak

Writing Compilers and Interpreters

A Modern Software Engineering Approach Using Java®

Third Edition

Ronald Mak

Writing Compilers and Interpreters: A Modern Software Engineering Approach Using Java®, Third Edition

This book is dedicated to all programmers who accept the challenge of writing very complex software successfully.

About the Author

Ronald Mak wrote the earlier editions of this very successful book, as well as *The Martian Principles for Successful Enterprise Systems: 20 Lessons Learned from NASA's Mars Exploration Rover Mission* (also published by Wiley) and *Java Number Cruncher: The Java Programmer's Guide to Numerical Computing* (Prentice Hall). He develops advanced software systems for organizations from startups to NASA. Currently a research staff member at the IBM Almaden Research Center, he also teaches compiler writing and software engineering at San José State University. He has degrees in the mathematical sciences and in computer science from Stanford University, and he lives in San José, CA with two cats.

Credits

Acknowledgments

Introduction

This book is about writing compilers and interpreters. The emphasis is on *writing* because this book writes a very large amount of code.

This is your book if you want to learn how to write an interpreter, a compiler, an interactive source-level debugger, and an integrated development environment (IDE) with a graphical user interface (GUI). All the code is in Java, which I explain in detail.

This book is not about the theory behind compiler writing. I leave that to the textbooks. If you want to learn the theory right now, then this is not your book. However, I hope that after working your way through this book's programs, you'll be inspired to learn about their theoretical underpinnings.

The first edition of this book used C as the implementation language, the second edition used C++, and this third edition uses Java. While I kept the basic organization, philosophy, and approach of the earlier editions, this edition is a complete rewrite.

What You'll Learn in this Book

The interpreter and the compiler that you learn to write in this book processes programs written in a high-level language. You'll write an interpreter that can execute programs. After you add the debugger, you'll be able to interact with the interpreter as it executes a program by setting breakpoints, displaying the call stack, viewing and modifying values of variables, and single-stepping the program's execution statement-by-statement. Add the IDE and you'll do all that with mouse clicks as you watch a program's execution animated on the screen. You'll learn to write a compiler that generates object code for the Java Virtual Machine (JVM). You'll be able to run compiled programs on multiple platforms. Of course, since the interpreter, compiler, debugger, and IDE are all written in Java, you'll be able to run them on multiple platforms, too.

The programming language of the source programs — the programs that your interpreter and compiler will process — is Pascal. I chose Pascal for several reasons. It's a real language, not a made-up one for this book. Pascal is a high-level procedure-oriented programming language that was very popular from the mid 1970s through the 1980s. The language has a relatively straightforward syntax, but it includes many of the language features that make compiler writing interesting, such as structured, user-definable data types, nested scopes, passing parameters by value and by reference, a full set of control statements, etc. Pascal continues to live today. You can download free Pascal interpreters and compilers from the Web to compare against the ones you'll write.[1]

[1] For example:

http://www.freepascal.org/.

A Software Engineering Approach

Compilers and interpreters are complex programs, and writing them successfully is *hard* work. To tackle the complexity, I take a strong software engineering approach in this book. Design patterns, Unified Modeling Language (UML) diagrams, and other modern object-oriented design practices make the code understandable and manageable.

Throughout the chapters, especially in the early ones, DESIGN NOTE sidebars point out design issues such as the use of a design pattern, or explain why I chose to architect the code a particular way.

The approach that I strongly believe in is: *Develop software incrementally*. At each step, get something to work. Build every step on working code from the previous step. Nearly every chapter of this book contains a major working program and often other shorter ones. Each chapter's programs build upon the ones from the previous chapters.

How the Book Is Organized

Chapter 1 is an introduction. Chapter 2 describes the framework for the compiler and the interpreter. It designs and tests this architectural foundation up front so that all the code in the remaining chapters can successfully build upon it. The next two chapters take care of some basic translation tasks, scanning (Chapter 3) and building a symbol table (Chapter 4).

The next several chapters build a working Pascal interpreter. To take the incremental development approach, these chapters iterate parsing and interpretation several times, with more of the Pascal language included in each iteration. Chapter 5 parses expressions and assignment statements and Chapter 6 interprets them. Chapter 7 parses the control statements and Chapter 8 interprets them. Chapter 9 parses declarations, Chapter 10 does type checking, and Chapter 11 parses procedures, functions, and entire Pascal programs. Chapter 12 completes the interpreter and executes entire Pascal programs.

The next two chapters build upon the working interpreter. Chapter 13 adds an interactive source-level debugger with which you communicate by typing commands on the command line. Chapter 14 wraps a GUI around this command-line debugger to create an IDE. You can skip these two chapters during your first time through the book without loss of continuity. But be sure to come back to them because they describe some really powerful software development tools.

The last part of the book develops the compiler by reusing much of the code from the earlier parts. Chapter 15 introduces the architecture of the Java Virtual Machine and Jasmin, the assembly language that the compiler will emit for the JVM.

Again, the incremental approach: Chapter 16 compiles programs, assignment statements, and expressions. Chapter 17 compiles procedures and function calls and string operations. Chapter 18 completes the compiler by compiling control statements and arrays and records.

The final Chapter 19 is a brief introduction to various compiler-writing topics that are not covered in the other chapters, such as code optimization and table-driven scanners and parsers.

Where to Get the Program Code

You can download all of the Java code developed in this book from the Web page at http://www.apropos-logic.com/wci/. There, you'll find instructions for how to download, install, compile, and run the programs. There are also some more Pascal test programs.

Table of Contents

Chapter 1

Introduction

This first chapter describes the goals of this book and its approach and presents an overview of compilers and interpreters.

Goals and Approach

This book teaches the basics of writing compilers and interpreters. Its goals are to show you how to design and develop

- A compiler written in Java for a major subset of Pascal, a high-level procedure-oriented programming language.[1] The compiler will generate code for the Java Virtual Machine (JVM).

  [1] See the preface for an explanation of how and why this book uses both Java and Pascal.

- An interpreter written in Java for the same Pascal subset that will include an interactive symbolic debugger.
- An integrated development environment (IDE) with a graphical user interface. The IDE will be a simplified version of full-featured IDEs such as what you would find with the open-source Eclipse or Borland's JBuilder. Nevertheless, it will include a source program editor and an interactive interface to set breakpoints, do single stepping, view and set variables values, and more.

These are very ambitious goals and successfully achieving them will be a major challenge! The right technical skills will provide *what* you need to do to compile a program into machine language or to interpret a program. Modern software engineering principles and good object-oriented design will show *how* to implement the code for the compiler or interpreter so that everything will work together correctly at the end. *Compilers and interpreters are large complex programs.* While you may be able to develop a small program successfully with only technical skills, anything as ambitious as a compiler or an interpreter will also require software engineering principles and object-oriented design. Therefore, this book emphasizes the necessary technical skills, modern software engineering principles, and good object-oriented design.

What Are Compilers and Interpreters?

The main purpose of a compiler or an interpreter is to translate a *source program* written in a high-level *source language*. Exactly what the source program is translated into is the subject of the next few paragraphs.

In this book, the source language will be a large subset of Pascal. In other words, you will compile and interpret Pascal programs. Since you will write the compiler and interpreter in Java, the *implementation language* is Java.

A Pascal compiler translates the *source* containing a Pascal program into the low-level machine language of a particular computer (or, more precisely, the machine language of the CPU). Usually, the source is in the form of a text file. If the compiler does its job correctly, the machine language version of the program will "say" the same thing as the original Pascal program.[2] The machine language is the *object language*, and the compiler generates *object code* (also called the *target code*) written in the machine language. A compiler's job is done after it generates the object code. Object code is often written to a file.[3]

[2] We'll rely on an intuitive notion of what it means for two programs to "say" the same thing. During execution, the two programs have the same behavior – they read the same input and produce the same output in the same sequence.
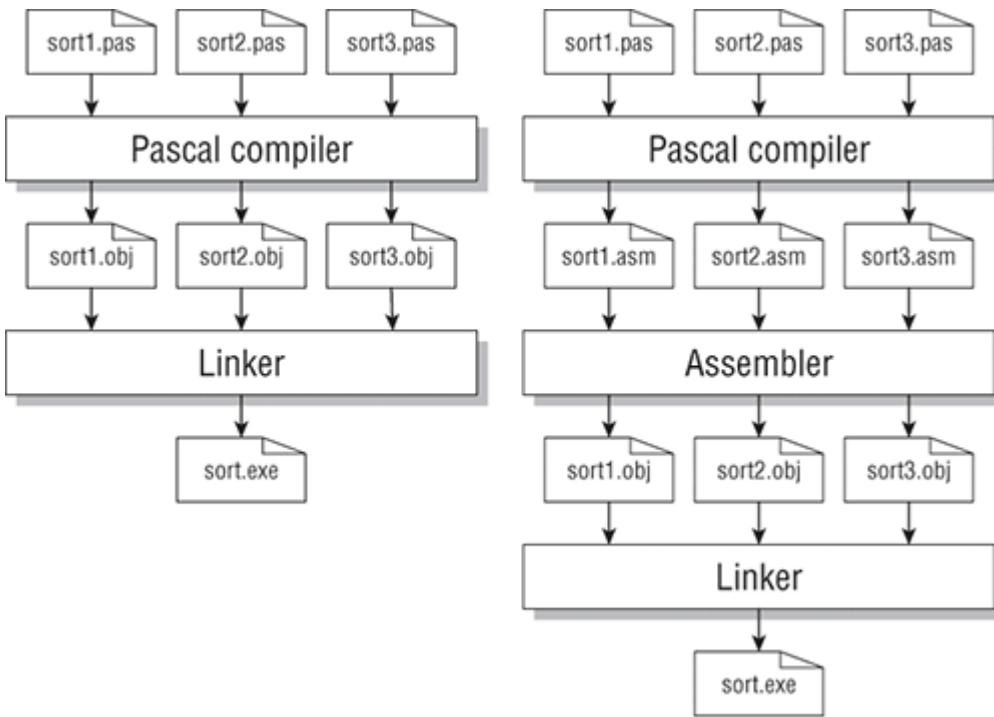
[3] Do not confuse the use of the word object in the terms object program and object code with its use in the term object oriented. These are entirely separate concepts – a program written in object-oriented language like Java or C++ or in a procedure-oriented language like Pascal would be compiled into an object program.

A program can consist of several source files, and the compiler generates a separate object file for each one. A utility program called a *linker* combines the contents of the one or more object files along with any needed runtime library routines into a single *object program* that the computer can load and execute. The library routines are often kept in precompiled object files.

Because machine language is not easily human-readable, a compiler can instead generate *assembly language* as the object language. Assembly language is one step up from machine language; there is usually a one-to-one mapping between each assembly instruction and machine language instruction. Assembly language is human-readable if you know the short mnemonics (such as ADD or LOAD) and understand the machine architecture. An *assembler* (itself a type of compiler) translates the assembly language to machine language.

Figure 1-1 summarizes the process of compiling one or more Pascal sources into an object program.

Figure 1-1: The diagram on the left shows a compiler translating a Pascal program consisting of three source files `sort1.pas`, `sort2.pas`, and `sort3.pas` into three corresponding machine-language object files `sort1.obj`, `sort2.obj`, and `sort3.obj`. The linker then combines the object files (along with any required runtime library routines) into the executable object program `sort.exe`. The diagram on the right shows a compiler translating the Pascal source files into assembly-language object files `sort1.asm`, `sort2.asm` and `sort3.asm`, which an assembler translates into the machine-language object files. The linker then produces the object program.

sort1.pas sort2.pas sort3.pas

Pascal compiler

sort1.obj sort2.obj sort3.obj

Linker

sort.exe

sort1.pas sort2.pas sort3.pas

Pascal compiler

sort1.asm sort2.asm sort3.asm

Assembler

sort1.obj sort2.obj sort3.obj

Linker

sort.exe

So how is an interpreter different from a compiler?

An interpreter does not generate an object program. Instead, after reading in a source program, it executes the program. This is analogous to what you'd do if you were handed a Pascal program and told to execute it by hand. You would read each statement in the proper sequence and mentally do what it says. You probably would keep track of the values of the program's variables on a sheet of scratch paper, and you'd write down each line of program output until you've completed the program execution. Essentially, you would have just done what a Pascal interpreter does. A Pascal interpreter reads in a Pascal program and executes it. There is no object program to generate and load. Instead, an interpreter translates a program into the actions that result from executing the program.

Comparing Compilers and Interpreters

How do you decide when to use an interpreter and when to use a compiler?

When you feed a source program into the interpreter, the interpreter takes over to check and execute the program. A compiler also checks the source program but instead generates object code. After running the compiler, you need to run the linker to generate the object program, and then you have to load the object program into memory to execute it. If the compiler generates an assembly language object code, you must also run an assembler. So, an interpreter definitely requires less effort to execute a program.

Interpreters can be more versatile than compilers. You can use Java to write a Pascal interpreter that runs on a Microsoft Windows-based PC, an Apple Macintosh, and a Linux box, so that the interpreter can execute Pascal source programs on any of those platforms. A compiler, however, generates object code for a particular computer. Therefore, even if you took a Pascal compiler originally written for the PC and made it run on the Mac, it would still generate object code for the PC. To make the compiler generate object code for the Mac, you would have to rewrite substantial portions of the compiler.[4]

[4] Later in this book, we'll get around this problem by making our compiler generate object code for the Java virtual machine. This virtual machine runs on various computer platforms.

What happens if the source program contains a logic error that doesn't show up until runtime, such as an attempt to divide by a variable whose value is zero?

Since an interpreter is in control when it is executing the source program, it can stop and tell you the line number of the offending statement and the name of the variable. It can even prompt you for some corrective action before resuming execution of the program, such as changing the value of the variable to something other than zero. An interpreter can include an interactive *source-level debugger*, also known as a *symbolic debugger*. *Symbolic* means the debugger allows you to use symbols from the source program, such as variable names.

On the other hand, the object program generated by a compiler and a linker generally runs by itself. Information from the source program such as line numbers and names of variables might not be present in the object program. When a runtime error occurs, the program may simply abort and perhaps print a message containing the address of the bad instruction. Then it's up to you to figure out the corresponding source statement and which variable's value was zero.

So when it comes to debugging, an interpreter is usually the way to go. Some compilers can generate extra information in the object code so that if a runtime error occurs, the object program can print out the offending statement line number or the variable name. You then fix the error, recompile, and rerun the program. Generating extra information in the object code can cause the object program to run slower than it otherwise could. This may induce you to turn off the debugging features when you're about to compile the final "production" version of your program.[5]

[5] This situation has been compared to wearing a life jacket when you're learning to sail on a lake, and then taking the jacket off when you're about to go out into the ocean.

Suppose you've successfully debugged your program, and now your primary concern is how fast it executes. Since a computer can execute a program in its native machine language at top speed, a compiled program can run orders of magnitude faster than an interpreted source program. A compiler is definitely the winner when it comes to speed. This is certainly true in the case of an optimizing compiler that knows how to generate especially efficient code.

So, whether you should use a compiler or an interpreter depends on what aspects of program development and execution are important. The best of both worlds would include an interpreter with an interactive symbolic debugger to use during program development and a compiler to generate machine language code for fast execution after you've debugged the program. Such are the goals of this book, since it teaches how to write both compilers and interpreters.

The Picture Gets a Bit Fuzzy

It used to be easier to explain the differences between an interpreter and a compiler. With the growing popularity of virtual machines, the picture gets a bit fuzzy.

A *virtual machine* is a program that simulates a computer. This program can run on different actual computer platforms. For example, the Java virtual machine (JVM) can run on a Microsoft Windows-based PC, an Apple Macintosh, a Linux system, and many others.

The virtual machine has its own virtual machine language, and the machine language instructions are interpreted by the actual computer that's running the virtual machine. So if you write a translator that translates a Pascal source program into virtual machine language that is interpreted, is the translator a compiler or an interpreter?

Rather than splitting hairs, let's agree for this book that if a translator translates a source program into machine language, whether for an actual computer or for a virtual machine, the translator is a compiler. A translator that executes the source program without first generating machine language is an interpreter.

Why Study Compiler Writing?

We've all learned to take compilers and interpreters mostly for granted. Because you need to concentrate on writing and debugging the program that you're developing, you don't even want to think about what the compiler is doing. You may notice the compiler only whenever you make a syntax error and the compiler flags it with an error message. You want to assume that if there are no syntax errors, the compiler will generate the correct code. If your program behaves badly at run time, you might be tempted to blame the compiler for generating bad code, but the vast majority of the time, you'll discover the error is actually in your program.

This is generally the situation if you're using one of the popular standard programming languages, such as Java or C++. Compilers, interpreters, and IDEs are all provided for you. End of story.

Recently, however, we've seen much activity in the development of new programming languages. Driving forces include the World Wide Web and new languages to accommodate developing web-based applications. Ever-increasing pressures to improve programmer productivity have also spurred the creation of languages that are well-tuned for specific application domains. You may very well find yourself one day inventing a scripting language to express algorithms or to control processes in your domain. If you invent a new language, you'll have to develop a compiler or an interpreter for it.

A compiler or an interpreter is a very interesting program in its own right. Each one is certainly not an insignificant program. As mentioned above, appropriate technical skills, modern software engineering principles, and good object-oriented design are required to develop them successfully. So alongside the intellectual satisfaction of learning how compilers and interpreters work, you can also appreciate the challenge of writing them.

Conceptual Design

To prepare for the next few chapters, let's examine the conceptual design of a compiler or an interpreter.

> ## Design Note
>
> The conceptual design of a program is a high-level view of its software architecture. The conceptual design includes the primary components of the program, how they're organized, and how they interact with each other. It does not necessarily say how these components will be implemented. Rather, it allows you to examine and understand the components first without worrying about how you're eventually going to develop them.

You can classify both compilers and interpreters as programming language translators. As explained above, a compiler translates a source program into machine language, and an interpreter translates the program into actions. Such a translator, as seen at the highest level, consists of a *front end* and a *back end*. Following the principle of software reuse, you'll soon see that a Pascal compiler and a Pascal interpreter can share the same front end, but they'll each have a different back end.

The front end of a translator reads the source program and performs the initial translation stage. Its primary components are the *parser*, the *scanner*, the *token*, and the *source*.
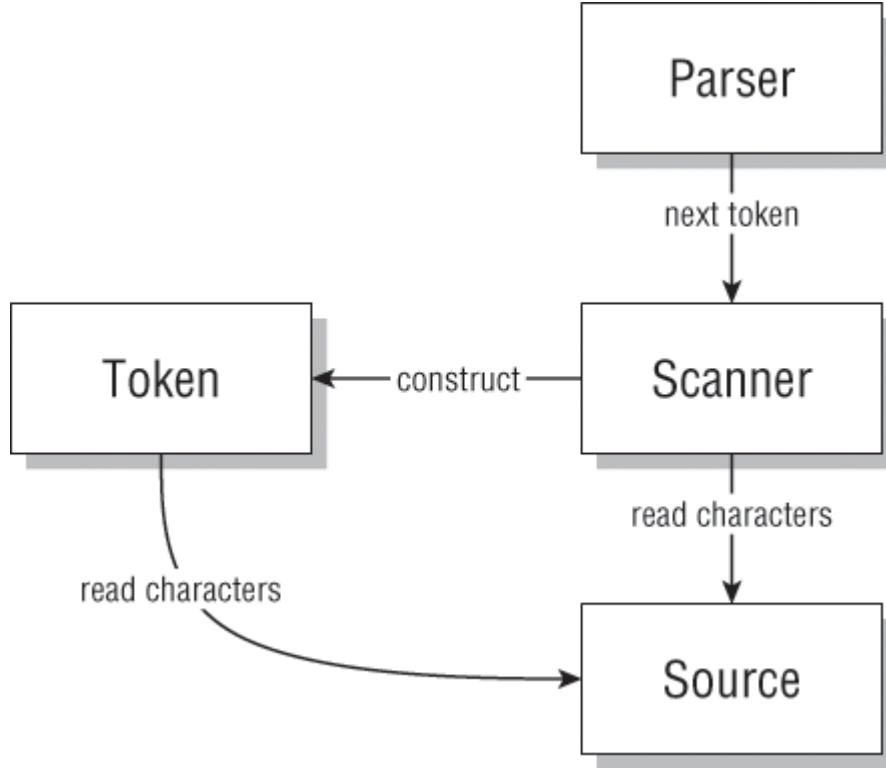
The parser controls the translation process in the front end. It repeatedly asks the scanner for the next token, and it analyzes the sequences of tokens to determine what high-level language elements it is translating, such as arithmetic expressions, assignment statements, or procedure declarations. The parser verifies that what it sees is syntactically correct as written in the source program; in other words, the parser detects and flags any syntax errors. What the parser does is called *parsing*, and the parser *parses* the source program to translate it.

The scanner reads the characters of the source program sequentially and constructs *tokens*, which are the low-level elements of the source language. For example, Pascal tokens include *reserved words* such as BEGIN, END, IF, THEN, and ELSE, *identifiers* that are names

of variables, procedures, and functions, and *special symbols* such as = := + - * and /. What the scanner does is called *scanning*, and the scanner *scans* the source program to break it apart into tokens.

Figure 1-2 shows the conceptual design of the front end of a compiler or an interpreter.

Figure 1-2: The conceptual design for the front end

In this figure, an arrow represents a command issued by one component to another. The parser tells the scanner to get the next token. The scanner reads characters from the source and constructs a new token. The token also reads characters from the source. (Chapter 3 explains why *both* the scanner and token components need to read characters from the source.)

A compiler ultimately translates a source program into machine language object code, so the primary component of its back end is a *code generator*. An interpreter executes the program, so the primary component of its back end is an *executor*.

If you want the compiler and the interpreter to share the same front end, their different back ends need a common intermediate interface with the front end. Recall that the front end performs the initial translation stage. The front end generates *intermediate code* and a *symbol table* in the *intermediate tier* that serve as the common interface.

Intermediate code is a predigested form of the source program that the back end can process efficiently. In this book, the intermediate code will be an in-memory tree data structure that represents the statements of the source program. The symbol table contains information about the symbols (such as the identifiers) contained in the source program. A compiler's back end processes the intermediate code and the symbol table to generate the machine language version of the source program. An interpreter's back end processes the intermediate code and the symbol table to execute the program.

To further software reuse, you can design the intermediate code and the symbol table structures to be *language independent*. In other words, you can use the same structures for different source languages. Therefore, the back end will also be language independent; when it processes these structures, it doesn't need to know or care what the source language was.
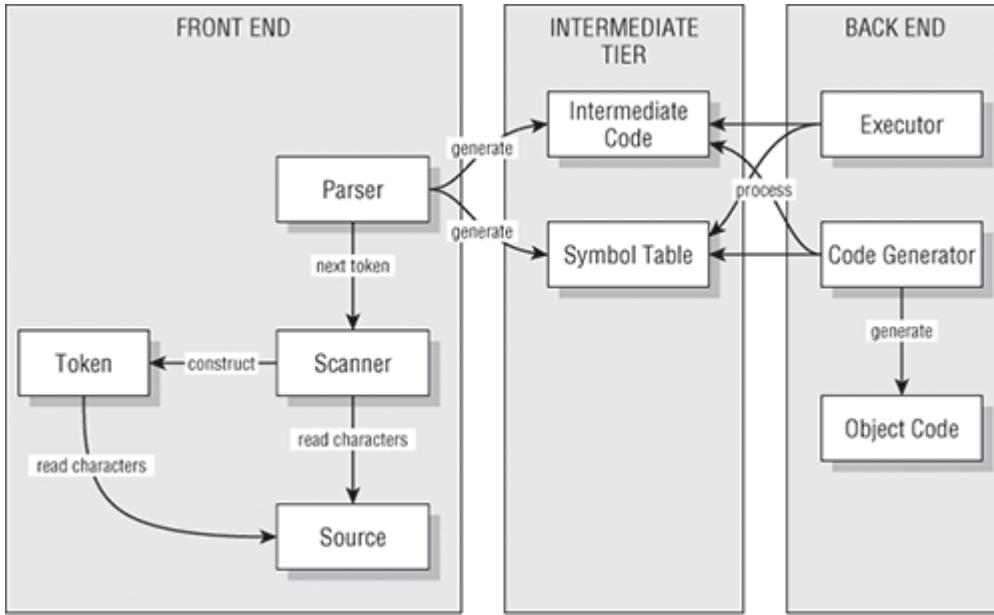
Figure 1-3 shows a more complete conceptual design of compilers and interpreters. If we design everything properly, only the front end needs to know which language the source programs are written in, and only the back end needs to distinguish between a compiler and an interpreter.

Start to flesh out this conceptual design by designing a framework for compilers and interpreters in Chapter 2. Chapter 3 is all about scanning. Build your first symbol table in Chapter 4, and in Chapter 5 generate some initial intermediate code. Begin the executor in Chapter 6 and develop it incrementally through Chapter 14, including the symbolic debugger and IDE. Code generation will have to wait until Chapter 16, after you've learned about the architecture of the JVM in Chapter 15.

Syntax and Semantics

The *syntax* of a programming language is its set of grammar rules that determine whether a statement or an expression is correctly written in that language. The language's *semantics* give meaning to a statement or an expression.

Figure 1-3: A more complete conceptual design

For example, Pascal's syntax tells us that

```
i := j + k
```

is a valid assignment statement. The language's semantics tells us that the statement says to add the current value of variable `j` and the current value of variable `k` and assign the sum as the new value of variable `i`.

A parser performs actions based on both the source language's syntax and semantics. Scanning the source program and extracting tokens are syntactic actions. Looking for the `:=` token after the target variable of an assignment statement is a syntactic action. Entering identifiers `i`, `j`, and `k` into the symbol table as variables, or looking them up in the symbol table, are semantic actions because the parser had to understand the meaning of the expression and the assignment to know that it needed to use the symbol table. Generating intermediate code that represents the assignment statement is a semantic action.

Syntactic actions occur only in the front end. Semantic actions occur in the front and back ends. Executing a program in the back end or generating object code for it requires knowing the meaning of its statements and so they consist of semantic actions. The intermediate code and the symbol table store semantic information.

Lexical, Syntax, and Semantic Analyses

Lexical analysis is the formal term for scanning, and thus a scanner can be called a *lexical analyzer*. *Syntax analysis* is the formal term for parsing, and a *syntax analyzer* is the parser. *Semantic analysis* involves checking that semantic rules aren't broken. An example is *type checking*, which ensures that the types of operands are consistent with their operators. Other operations of semantic analysis are building the symbol table and generating the intermediate code.

Chapter 2

Framework I: Compiler and Interpreter

This chapter advances you from the previous chapter's conceptual design to an initial implementation. You'll first build a flexible framework that can support both a compiler and an interpreter. Then you'll integrate rudimentary compiler and interpreter components into the framework. Finally, you'll run end-to-end tests to verify the framework and the components.

Goals and Approach

This chapter's design approach may at first appear to be overly fussy. Indeed, at the end, there may be a larger number of Java classes than you might have otherwise expected. But you're going to adhere to proven software engineering principles and good object-oriented design to create a framework upon which you can build both a compiler and an interpreter.

As stated in the conceptual design, the compiler and the interpreter should share as many components as possible, with only the back end components different between the two. In this chapter, you'll build a flexible framework and initially populate it with compiler and interpreter components that are greatly simplified or "stubbed out" altogether. Nevertheless, there'll be enough there to verify that you've designed the framework properly, that the components are well-integrated, and that they all can work together. The success criteria will be to run simple end-to-end threads of execution that go from the common front end through to the compiler and interpreter back ends, then incrementally develop the components further in the following chapters.

The goals for this chapter are:

- A source language-independent framework that can support both compilers and interpreters.
- Initial Pascal source language-specific components integrated into the front end of the framework.
- Initial compiler and interpreter components integrated into the back end of the framework.
- Simple end-to-end runs that exercise the components by generating source program listings from the common front end and messages from the compiler or interpreter back end.

Design Note

Whenever you develop a complex program such as a compiler or an interpreter, key first steps for success are:

1. Design and implement a proper framework.

2. Develop initial components that are well-integrated with the framework and with each other.

3. Test the framework and the component integration by running simple end-to-end threads of execution.

Early component integration is critical, even if you've simplified or stubbed out the initial components. Test your framework and components and get them working together as early as possible. The framework and the initial components then form the basis upon which you can do further development. Development should proceed incrementally and the code should continue to work (with more functionality) after each increment. You should always be building on code that already works.
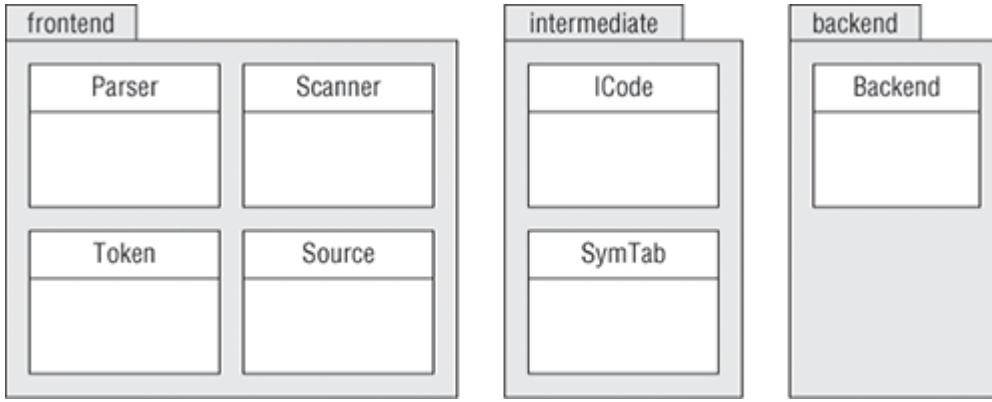
Language-Independent Framework Components

Based on the conceptual design, the framework consists of three packages: `frontend`, `intermediate`, and `backend`.

Framework components are source language-independent interfaces and classes that define the framework. Some of the classes can be abstract. Once the framework components are in place, you can develop Pascal language-specific implementations of the abstract classes.

Figure 2-1 shows the framework components using UML package and class diagrams.

Figure 2-1: These language-independent components contained in the `frontend`, `intermediate`, and `backend` packages together define a framework that will support both a compiler and an interpreter.
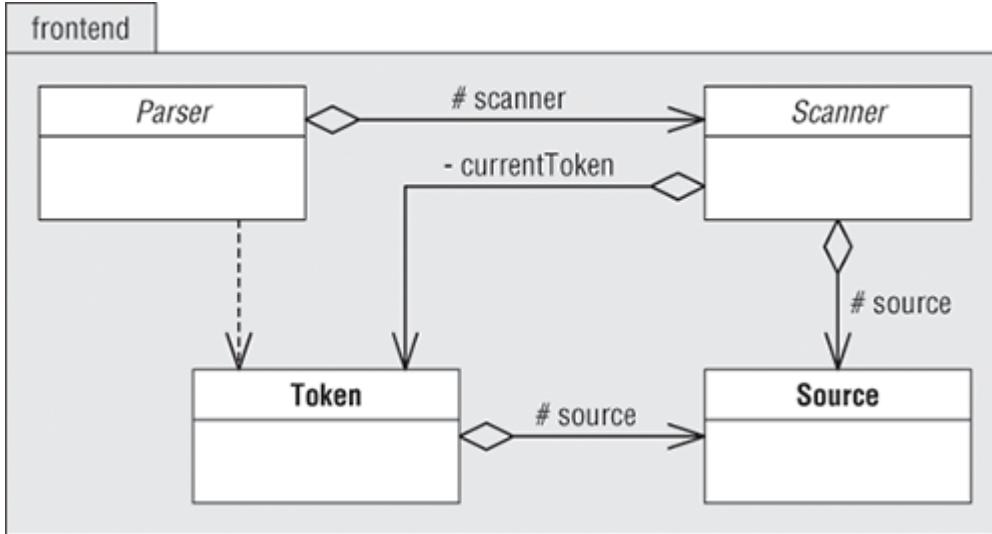
Design Note

The Unified Modeling Language (UML) is an industry-standard graphical language for representing the design of object-oriented software architectures and processes. Its various types of diagrams can model the static relationships of a program's architectural components and the dynamic runtime behavior of these components.

Front End

In the `frontend` package, the language-independent `Parser`, `Scanner`, `Source`, and `Token` classes represent the framework components. The framework classes force you to think hard about the responsibilities of each front end component and how they'll interact with each other, regardless of the source language. The UML class diagram in Figure 2-2 shows their relationships.
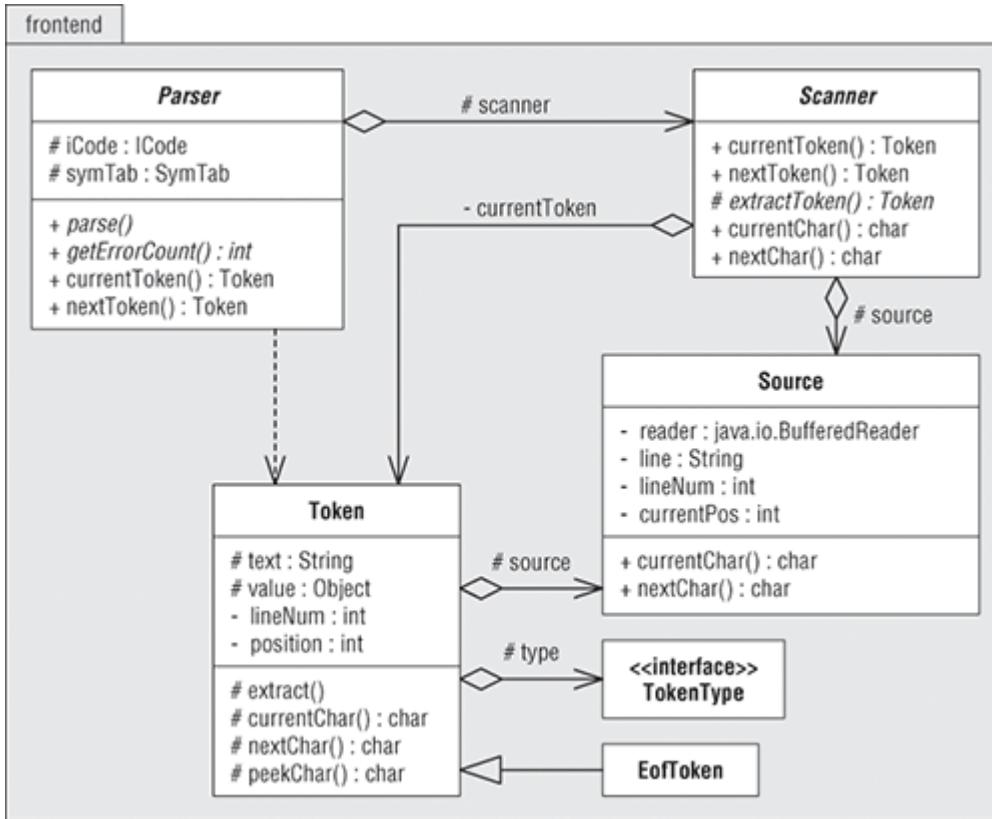
Figure 2-2: The `Parser`, `Scanner`, `Source`, and `Token` classes constitute the framework components of the `frontend` package.

The `Parser` and `Scanner` classes are abstract; language-specific subclasses will implement their abstract methods. The parser and the scanner are closely related. The parser "owns" a scanner by using its protected `scanner` field to keep a reference to the scanner. The parser requests tokens from its scanner, and so it has a dependency on tokens. The scanner owns the current token using its private `currentToken` field. It owns the source via the protected `source` field; it passes the source reference to each token it constructs. Each token then also owns that source via its protected `source` field. During its construction, a token reads characters from the source.

The UML class diagram in Figure 2-3 shows the four front end framework classes in greater detail. It shows fields and methods and other front end classes and interfaces. For example, each token has a token type represented by the interface `TokenType`, and `EofToken` is a subclass of `Token`.

Figure 2-3: The `frontend` framework classes in greater detail

According to the conceptual design, the parser controls the translation process. It parses the source program, and so the `Parser` class has an abstract `parse()` method; language-specific implementations of this method will repeatedly ask the scanner for the next token. The parser's `currentToken()` and `nextToken()` convenience methods in turn call the scanner's `currentToken()` and `nextToken()` methods, respectively.[1] A language-specific implementation of the abstract `getErrorCount()` method will return the number of syntax errors.

---

[1] The parser methods `currentToken()` and `nextToken()` are "convenient" because they save us the trouble of writing the longer method calls `scanner.currentToken()` and `scanner.nextToken()`.

Design Note

In a UML class diagram, an arrow with an open arrowhead represents a reference or dependency by one class to another. A dashed arrow (such as from `Parser` to `Token`) is a transient reference that exists only during a method call (such as the parser's `nextToken()` method which returns a `Token` object). A solid arrow with a hollow diamond at the owner's end (such as from `Parser` to `Scanner`) indicates that one class "owns" or "aggregates" another using a reference that lasts the lifetime of an object. The name of the field that holds the reference labels the arrow (for example, the parser uses its `scanner` field to maintain a reference to its scanner).

A solid arrow with a closed hollow arrowhead (such as from `EofToken` to `Token`) points from a subclass to its superclass.

Below the class name, a class diagram can optionally include sections for the fields and for the methods. Field names that are arrow labels do not appear again inside the field section. A character before each field or method name indicates access control:

+ public

- private

protected

~ package

A colon separates each field name or method name from the field type or the return type, respectively. To save space, class diagrams usually don't show constructors and field getter and setter methods.

The name of an abstract class (such as `Parser` and `Scanner`) is in italics. The name of an abstract method is also in italics.

The scanner constructs tokens extracted from the source program. Language-specific implementations of the `Scanner` class's abstract `extractToken()` method will read characters from the `Source` in order to construct the tokens of a particular language. The scanner's convenience methods `currentChar()` and `nextChar()` call the corresponding methods of the `Source` class. (There will be more shortly about "current" and "next" for source characters.)

Fields of the `Token` class store useful information about a token, including its type, text string, value, and location (line number and position) in the source program. It also has `currentChar()` and `nextChar()` methods that will in turn call the `currentChar()` and `nextChar()` methods of the `Source` class. Token types are language-specific, so for now, the `TokenType` interface serves as a placeholder.

Later, you'll define language-specific token subclasses to represent the tokens of a particular language. For now, only define the language-independent `EofToken` subclass to represent the end of the source file. Using token subclasses will keep the scanner code modular, since you'll need different algorithms to construct the various types of tokens.

Parser

shows the key methods of the abstract `Parser` framework class. A language-specific subclass will provide the implementations of the `parse()` and `getErrorCount()` methods to parse the source program and to return the number of syntax errors, respectively. As mentioned above, the `Parser` class implements the `currentToken()` and `nextToken()` convenience methods that delegate to the corresponding scanner methods which return the current or the next token, respectively.

Listing 2-1: Key methods of the abstract `Parser` framework class

```
package wci.frontend;

import wci.intermediate.ICode;
import wci.intermediate.SymTab;

/**
 * <h1>Parser</h1>
 *
 * <p>A language-independent framework class.  This abstract parser class
 * will be implemented by language-specific subclasses.</p>
 */
public abstract class Parser
{
    protected static SymTab symTab;  // generated symbol table

    static {
        symTab = null;
    }

    protected Scanner scanner; // scanner used with this parser
    protected ICode iCode;     // intermediate code generated by this parser

    /**
     * Constructor.
     * @param scanner the scanner to be used with this parser.
     */
    protected Parser(Scanner scanner)
    {
        this.scanner = scanner;
        this.iCode = null;
    }

    /**
     * Parse a source program and generate the intermediate code and the
     * symbol table. To be implemented by a language-specific parser
     * subclass.
     * @throws Exception if an error occurred.
     */
    public abstract void parse()
        throws Exception;

    /**
     * Return the number of syntax errors found by the parser.
     * To be implemented by a language-specific parser subclass.
     * @return the error count.
     */
    public abstract int getErrorCount();

    /**
     * Call the scanner&apos;s currentToken() method.
     * @return the current token.
     */
    public Token currentToken()
    {
        return scanner.currentToken();
    }

    /**
     * Call the scanner&apos;s nextToken() method.
     * @return the next token.
     * @throws Exception if an error occurred.
     */
    public Token nextToken()
        throws Exception
    {
        return scanner.nextToken();
    }
}
```

Because there will be only one symbol table in the front end, class `Parser` implements `symTab` as a class field. The static block will initialize it once before any `Parser` objects are created.

Source

shows the key methods of the `Source` framework class.

Listing 2-2: Key methods of the `Source` framework class

```
package wci.frontend;

import java.io.BufferedReader;
import java.io.IOException;

/**
 * <h1>Source</h1>
 *
 * <p>The framework class that represents the source program.</p>
 */
public class Source
{
    public static final char EOL = &apos;\n&apos;;      // end-of-line character
    public static final char EOF = (char) 0;  // end-of-file character
```

```java
private BufferedReader reader;          // reader for the source program
private String line;                    // source line
private int lineNum;                    // current source line number
private int currentPos;                 // current source line position

/**
 * Constructor.
 * @param reader the reader for the source program
 * @throws IOException if an I/O error occurred
 */
public Source(BufferedReader reader)
    throws IOException
{
    this.lineNum = 0;
    this.currentPos = -2;  // set to -2 to read the first source line
    this.reader = reader;
}

/**
 * Return the source character at the current position.
 * @return the source character at the current position.
 * @throws Exception if an error occurred.
 */
public char currentChar()
    throws Exception
{
    // First time?
    if (currentPos == -2) {
        readLine();
        return nextChar();
    }

    // At end of file?
    else if (line == null) {
        return EOF;
    }

    // At end of line?
    else if ((currentPos == -1) || (currentPos == line.length())) {
        return EOL;
    }

    // Need to read the next line?
    else if (currentPos > line.length()) {
        readLine();
        return nextChar();
    }

    // Return the character at the current position.
    else {
        return line.charAt(currentPos);
    }
}

/**
 * Consume the current source character and return the next character.
 * @return the next source character.
 * @throws Exception if an error occurred.
 */
public char nextChar()
    throws Exception
{
    ++currentPos;
    return currentChar();
}

/**
 * Return the source character following the current character without
 * consuming the current character.
 * @return the following character.
 * @throws Exception if an error occurred.
 */
public char peekChar()
    throws Exception
{
    currentChar();
    if (line == null) {
        return EOF;
    }

    int nextPos = currentPos + 1;
    return nextPos < line.length() ? line.charAt(nextPos) : EOL;
}

/**
 * Read the next source line.
 * @throws IOException if an I/O error occurred.
 */
private void readLine()
    throws IOException
{
    line = reader.readLine();  // null when at the end of the source
    currentPos = -1;

    if (line != null) {
        ++lineNum;
    }
}

/**
 * Close the source.
 * @throws Exception if an error occurred.
 */
public void close()
    throws Exception
{
    if (reader != null) {
        try {
            reader.close();
        }
        catch (IOException ex) {
```

```
                    ex.printStackTrace();
                    throw ex;
                }
            }
        }
    }
```

The constructor's argument is a reference to the `BufferedReader` object that the `Source` object will own. You'll see later that it is easy to create a `BufferedReader` object from a source file. You can also create a `BufferedReader` object from other types of objects such as strings. So `BufferedReader` is actually an abstraction; you don't want the `Source` class to care where the text of the source program comes from.

Method `currentChar()` does most of the work. In the very first call to the method, it calls method `readLine()` to read the first line, and `currentChar()` returns the first character of the line. In subsequent calls to the method, if the current position is at the end of the source line, it returns the special `EOL` (end of line) character. If it's already beyond the end of line, `currentChar()` again calls `readLine()` and returns the first character of the new line. At the end of the source, `line` will be null and so `currentChar()` returns a special `EOF` (end of file) character. In all other cases, the method simply returns the character at the current source line position `currentPos`.

Method `nextChar()` returns the next source character by first advancing the current line position `currentPos` and then calling method `currentChar()`.

Suppose the source line contains `ABCDE` and `currentPos` is 0 (the first line position). Then in the following sequence of calls to `currentChar()` and `nextChar()`, each call returns the character as shown:

```
currentChar() ⇒ 'A'
nextChar()    ⇒ 'B'
nextChar()    ⇒ 'C'
nextChar()    ⇒ 'D'
currentChar() ⇒ 'D'
currentChar() ⇒ 'D'
nextChar()    ⇒ 'E'
nextChar()    ⇒ EOL
```

`nextChar()` "consumes" the current character (by incrementing `currentPos` to the next character), but `currentChar()` doesn't. Sometimes you'll call `nextChar()` simply to consume the current character without storing the character it returns. You'll see below and in the next chapter how to use both methods.

Method `peekChar()` "looks ahead" one source character beyond the current character without consuming the current character. In the next chapter, this method will distinguish between the single Pascal token `"3.14"` and the three Pascal tokens `"3..14"`. Note that `peekChar()` always returns the `EOL` character whenever the current position is at the end of the source line or beyond instead of reading the next line. This simplification won't cause trouble given the way you'll use the method.

Besides updating field `line`, method `readLine()` increments `lineNum` and resets `currentPos` to 0.

Scanner

Listing 2-3 shows the abstract `Scanner` framework class. A language-specific subclass will implement the `extractToken()` method. The parser calls method `nextToken()`, which calls `extractToken()` to set and return the value of private field `currentToken`. The convenience methods `currentChar()` and `nextChar()` call the corresponding methods of the `Source` class.

Listing 2-3: The abstract `Scanner` framework class

```
package wci.frontend;

/**
 * <h1>Scanner</h1>
 *
 * <p>A language-independent framework class.  This abstract scanner class
 * will be implemented by language-specific subclasses.</p>
 */
public abstract class Scanner
{
    protected Source source;      // source
    private Token currentToken;   // current token

    /**
     * Constructor
     * @param source the source to be used with this scanner.
     */
    public Scanner(Source source)
    {
        this.source = source;
    }

    /**
     * @return the current token.
     */
    public Token currentToken()
    {
        return currentToken;
    }

    /**
     * Return next token from the source.
     * @return the next token.
     * @throws Exception if an error occurred.
     */
    public Token nextToken()
        throws Exception
    {
        currentToken = extractToken();
        return currentToken;
    }

    /**
     * Do the actual work of extracting and returning the next token from the
     * source. Implemented by scanner subclasses.
     * @return the next token.
     * @throws Exception if an error occurred.
     */
    protected abstract Token extractToken()
        throws Exception;

    /**
```

```
     * Call the source&apos;s currentChar() method.
     * @return the current character from the source.
     * @throws Exception if an error occurred.
     */
    public char currentChar()
        throws Exception
    {
        return source.currentChar();
    }

    /**
     * Call the source&apos;s nextChar() method.
     * @return the next character from the source.
     * @throws Exception if an error occurred.
     */
    public char nextChar()
        throws Exception
    {
        return source.nextChar();
    }
}
```

Token

Listing 2-4 shows the key methods of the `Token` framework class.

Listing 2-4: Key methods of the `Token` framework class

```
package wci.frontend;

/**
 * <h1>Token</h1>
 *
 * <p>The framework class that represents a token returned by the scanner.</p>
 */
public class Token
{
    protected TokenType type;  // language-specific token type
    protected String text;     // token text
    protected Object value;    // token value
    protected Source source;   // source
    protected int lineNum;     // line number of the token&apos;s source line
    protected int position;    // position of the first token character

    /**
     * Constructor.
     * @param source the source from where to fetch the token&apos;s characters.
     * @throws Exception if an error occurred.
     */
    public Token(Source source)
        throws Exception
    {
        this.source = source;
        this.lineNum = source.getLineNum();
        this.position = source.getPosition();

        extract();
    }

    /**
     * Default method to extract only one-character tokens from the source.
     * Subclasses can override this method to construct language-specific
     * tokens.  After extracting the token, the current source line position
     * will be one beyond the last token character.
     * @throws Exception if an error occurred.
     */
    protected void extract()
        throws Exception
    {
        text = Character.toString(currentChar());
        value = null;

        nextChar();  // consume current character
    }

    /**
     * Call the source&apos;s currentChar() method.
     * @return the current character from the source.
     * @throws Exception if an error occurred.
     */
    protected char currentChar()
        throws Exception
    {
        return source.currentChar();
    }
    /**
     * Call the source&apos;s nextChar() method.
     * @return the next character from the source after moving forward.
     * @throws Exception if an error occurred.
     */
    protected char nextChar()
        throws Exception
    {
        return source.nextChar();
    }

    /**
     * Call the source&apos;s peekChar() method.
     * @return the next character from the source without moving forward.
     * @throws Exception if an error occurred.
     */
    protected char peekChar()
        throws Exception
    {
        return source.peekChar();
    }
}
```

According to the conceptual design, the scanner constructs tokens and returns them to the parser. Since `TokenType` is an interface, you can set a token's `type` field to a language-specific value. The next chapter will show how the scanner determines the type of the next token to construct based on the current source character, which will become the initial character of the token. For example, if the initial character is a digit, the next token is a number. If the initial character is a letter, the next token is either an identifier or a

reserved word. Since you'll represent the different types of tokens using subclasses of the `Token` class, the scanner will call the constructor of the appropriate token subclass based on the initial character.

The constructor calls method `extract()` to do the actual work of constructing the token. As its name implies, the method extracts the token from the source by reading characters. Token subclasses will override this method to implement algorithms that are specific to language-specific token types. The `Token` class provides a default implementation that constructs a one-character token. With few exceptions, implementations of method `extract()` will consume the source characters of the token and leave the current source line position *one beyond* the last token character.

Before calling method `extract()`, the constructor sets the line number of the source line containing the token text, and the beginning position of the token text within that line. For example, the text string of the reserved word `BEGIN` can be `"begin"` (Pascal is not case-sensitive). If the text `"begin"` is in positions 11 through 15, the beginning position will be 11, and upon returning from `extract()`, the current position will be 16.

Some tokens will have values. For example, a numeric token may have the text `"3.14159"` and a floating-point value that is an approximation of *pi*.

Like class `Scanner`, class `Token` also implements the `currentChar()` and `nextChar()` methods so that they call the corresponding methods of the source object. The class also implements method `peekChar()` similarly.

Listing 2-5 shows the `TokenType` marker interface.[2] Language-specific token types will implement this interface.

[2] A *marker interface* does not define any methods, since its purpose is to "tag" any class that implements that interface. For example, any class that implements the `TokenType` interface is tagged to be a token type, even though the interface does not imply any specific behavior.

Listing 2-5: Interface `TokenType`

```
package wci.frontend;

public interface TokenType
{
}
```

Listing 2-6 shows the language-independent `EofToken` subclass. Since it represents the end of the source, it overrides the `extract()` method to do nothing.

Listing 2-6: Class `EofToken`

```
package wci.frontend;

/**
 * <h1>EofToken</h1>
 *
 * <p>The generic end-of-file token.</p>
 */
public class EofToken extends Token
{
    /**
     * Constructor.
     * @param source the source from where to fetch subsequent characters.
     * @throws Exception if an error occurred.
     */
    public EofToken(Source source)
        throws Exception
    {
        super(source);
    }

    /**
     * Do nothing.  Do not consume any source characters.
     * @param source the source from where to fetch the token&apos;s characters.
     * @throws Exception if an error occurred.
     */
    protected void extract(Source source)
        throws Exception
    {
    }
}
```

You'll develop subclasses for the various Pascal tokens in Chapter 3.

Messages

While it is translating a source program, the parser may need to report some status information, such as an error message whenever it finds a syntax error. However, you don't want the parser to worry about where it should send the message or what the recipient does with it.

Similarly, whenever the source component reads a new line, it can send a message containing the text of the line and the line number. The recipient may want to use these messages to produce a source listing, but you don't want the source component to care about that.
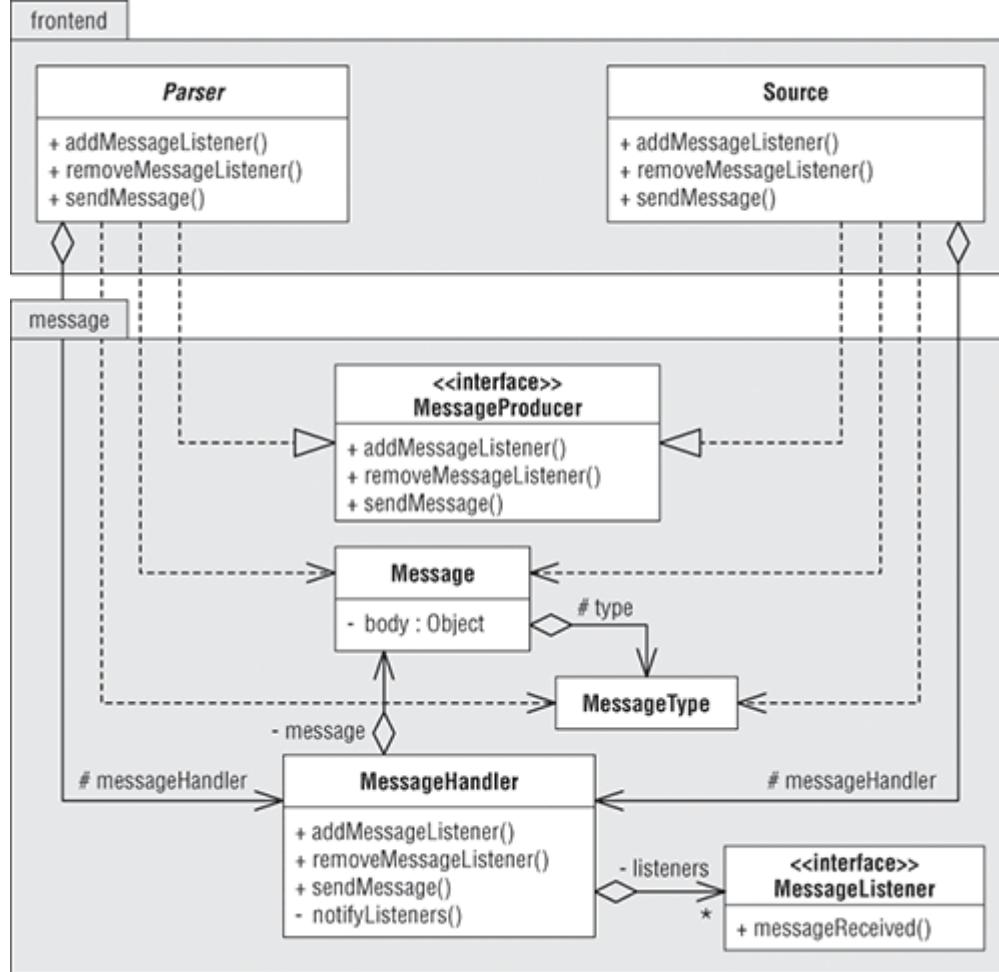
> ### Design Note
>
> Keep the senders of messages (the parser and the source) loosely coupled to the recipients of the messages. Two components are loosely coupled to each other when you minimize the dependencies between them, and then it is much easier to make changes to one component without adversely affecting the other. In a large complex application, loose coupling allows you to develop components independently and in parallel, such as by a team of programmers.

Figure 2-4 expands upon Figure 2-3 and shows that the `message` package contains messaging interfaces and classes that the `Parser` and `Source` classes use. Both classes implement interface `MessageProducer` and each owns a `MessageHandler`. Both classes also refer to the `Message` and `MessageType` classes.

Design Note

In UML class diagrams, a dashed arrow with a closed hollow arrowhead (such as from `Parser` to `MessageProducer` and from `Source` to `MessageProducer`) points from a class to an interface that the class implements.

Figure 2-4: The `message` package

Both the `Parser` and `Source` classes implement the `MessageProducer` interface and therefore, each class must define the `addMessageListener()`, `removeMessageListener()`, and `sendMessage()` methods. Instead, any object that wants to receive (i.e., listen to) the parser or source messages can subscribe to them by calling the parser's or the source's `addMessageListener()` method, respectively. An object that no longer wants to listen can call the `removeMessageListener()` method. The parser or the source can each have multiple message listeners, and whenever it has a message, it will send that message to each and every one of them.[3]

---

[3] "Sending" a message is used in the generic sense. What you actually do is pass the message string as the argument of a call to method `messageReceived()`, which is defined by interface `MessageListener`.

A message producer can own a `MessageHandler` object, and each of its `addMessageListener()`, `removeMessageListener()`, and `sendMessage()` methods calls the corresponding message handler method. In other words, the message producer *delegates* these methods to its message handler.

Each time it reads the source reads new source line, its `readLine()` method calls the message handler's `sendMessage()` method to send out a `SOURCE_LINE` message containing the line number and the text of the source line to all listeners. *By convention*, the format of this message is:

SOURCE_LINE Message	
`lineNum`	source line number
`line`	text of source line

All `SOURCE_LINE` message listeners must follow this convention.

Listing 2-7 shows the `MessageProducer` interface.

Listing 2-7: Interface `MessageProducer`

```
package wci.message;

public interface MessageProducer
```

```
{
    /**
     * Add a listener to the listener list.
     * @param listener the listener to add.
     */
    public void addMessageListener(MessageListener listener);

    /**
     * Remove a listener from the listener list.
     * @param listener the listener to remove.
     */
    public void removeMessageListener(MessageListener listener);

    /**
     * Notify listeners after setting the message.
     * @param message the message to set.
     */
    public void sendMessage(Message message);
}
```

[Listing 2-8](#) expands upon [Listing 2-1](#) to show that class `Parser` implements interface `MessageProducer`, owns a `MessageHandler`, and implements methods `addMessageListener()`, `removeMessageListener()`, and `sendMessage()`.

[Listing 2-8:](#) Class `Parser` implements interface `MessageProducer`

```
public abstract class Parser implements MessageProducer
{
    protected static SymTab symTab;                  // generated symbol table
    protected static MessageHandler messageHandler;  // message handler delegate

    static {
        symTab = null;
        messageHandler = new MessageHandler();
    }

    ...

    /**
     * Add a parser message listener.
     * @param listener the message listener to add.
     */
    public void addMessageListener(MessageListener listener)
    {
        messageHandler.addListener(listener);
    }

    /**
     * Remove a parser message listener.
     * @param listener the message listener to remove.
     */
    public void removeMessageListener(MessageListener listener)
    {
        messageHandler.removeListener(listener);
    }

    /**
     * Notify listeners after setting the message.
     * @param message the message to set.
     */
    public void sendMessage(Message message)
    {
        messageHandler.sendMessage(message);
    }
}
```

Similarly, class `Source` implements interface `MessageProducer`. [Listing 2-9](#) shows its `readLine()` method, which sends a `SOURCE_LINE` message.

[Listing 2-9:](#) Method `readLine()` in class `Source`, which sends a `SOURCE_LINE` message

```
    /**
     * Read the next source line.
     * @throws IOException if an I/O error occurred.
     */
    private void readLine()
        throws IOException
    {
        line = reader.readLine();  // null when at the end of the source
        currentPos = 0;

        if (line != null) {
            ++lineNum;
        }

        // Send a source line message containing the line number
        // and the line text to all the listeners.
        if (line != null) {
            sendMessage(new Message(SOURCE_LINE,
                                    new Object[] {lineNum, line}));
        }
    }
```

All classes that subscribe to messages must implement the `MessageListener` interface. See [Listing 2-10](#). Each time a message producer produces a new message, it notifies all its message listeners by calling each listener's `messageReceived()` method, passing the message as an argument.

[Listing 2-10:](#) Interface `MessageListener`

```
package wci.message;

public interface MessageListener
{
    /**
     * Called to receive a message sent by a message producer.
     * @param message the message that was sent.
     */
    public void messageReceived(Message message);
}
```

Listing 2-11 shows the constructor of class `Message` in package `message`. This is the format for all messages sent to listeners. Each message has a type and a generic body, and the class provides a getter method for each field (not shown).

Listing 2-11: The `Message` class constructor

```
package wci.message;

/**
 * <h1>Message</h1>
 *
 * <p>Message format.</p>
 */
private MessageType type;
    private Object body;

    /**
     * Constructor.
     * @param type the message type.
     * @param body the message body.
     */
    public Message(MessageType type, Object body)
    {
        this.type = type;
        this.body = body;
    }
}
```

Listing 2-12 shows the `MessageType` enumerated type that represents the various types of messages.

Listing 2-12: Enumerated type `MessageType`

```
package wci.message;

public enum MessageType
{
    SOURCE_LINE, SYNTAX_ERROR,
    PARSER_SUMMARY, INTERPRETER_SUMMARY, COMPILER_SUMMARY,
    MISCELLANEOUS, TOKEN,
    ASSIGN, FETCH, BREAKPOINT, RUNTIME_ERROR,
    CALL, RETURN,
}
```

Listing 2-13 shows the delegate class `MessageHandler`. As you saw above, a message producer class can create a message handler and delegate work to it. The message handler maintains the list of message listeners. The message producer calls its `sendMessage()` method with a new message. Private method `notifyListeners()` sends the message to all listeners by calling each one's `messageReceived()` method.

Listing 2-13: Class `MessageHandler`

```
package wci.message;

import java.util.ArrayList;

/**
 * <h1>MessageHandler</h1>
 *
 * <p>A helper class to which message producer classes delegate the task of
 * maintaining and notifying listeners.</p>
 */
public class MessageHandler
{
    private Message message;                 // message
    private ArrayList<MessageListener> listeners;  // listener list

    /**
     * Constructor.
     */
    public MessageHandler()
    {
        this.listeners = new ArrayList<MessageListener>();
    }

    /**
     * Add a listener to the listener list.
     * @param listener the listener to add.
     */
    public void addListener(MessageListener listener)
    {
        listeners.add(listener);
    }

    /**
     * Remove a listener from the listener list.
     * @param listener the listener to remove.
     */
    public void removeListener(MessageListener listener)
    {
        listeners.remove(listener);
    }

    /**
     * Notify listeners after setting the message.
     * @param message the message to set.
     */
    public void sendMessage(Message message)
    {
        this.message = message;
        notifyListeners();
    }

    /**
     * Notify each listener in the listener list by calling the listener&apos;s
     * messageReceived() method.
     */
```

```
    private void notifyListeners()
    {
        for (MessageListener listener : listeners) {
            listener.messageReceived(message);
        }
    }
}
```

Design Note

The `MessageProducer` and `MessageListener` interfaces and the `MessageHandler` class together implement the Observer Design Pattern. This pattern allows message producers and message listeners to remain loosely coupled.
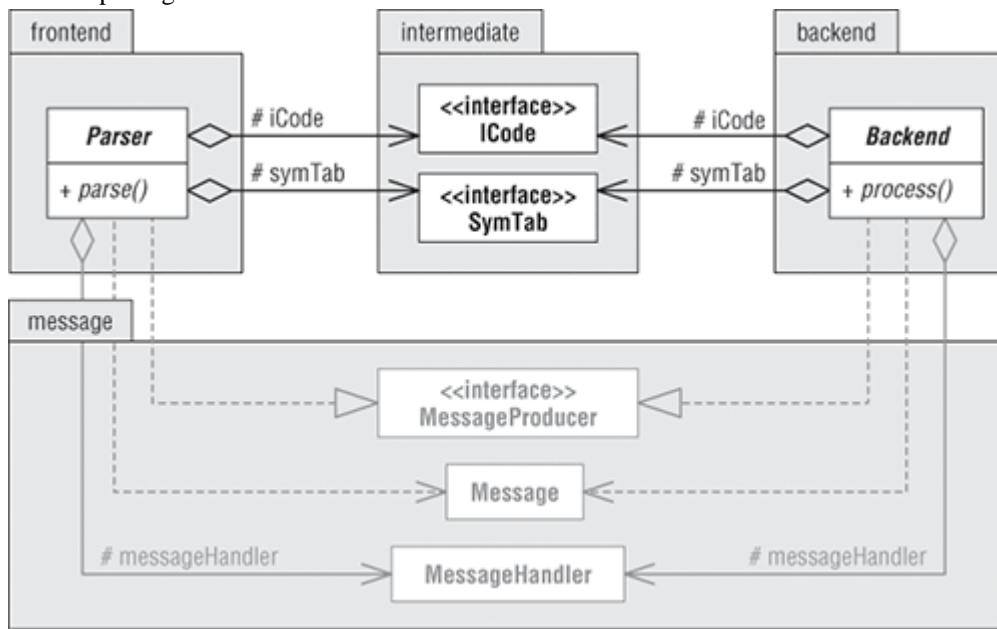
Loose coupling in this case means that a producer's responsibilities are limited to generating messages and notifying the listeners. The producer doesn't need to care who the listeners are or what they do with the messages. Without any code changes, it can add or remove listeners and accommodate any type of new listener that implements the `MessageListener` interface. Changes to the producers or to the listeners will not affect each other, as would be the case if they were tightly coupled.

A message producer class can use the `MessageHandler` helper class to do the work of maintaining and notifying its listeners. This is an example of delegation, a software engineering technique where one class asks another class to handle some task. Delegation also limits a class's responsibilities and supports loose coupling, and the delegate (the `MessageHandler` class in this case) can be used by other classes. This is more flexible than implementing the task in a superclass and forcing the producer classes to extend the superclass. In general, favor composition (with a delegate) over inheritance.

Intermediate Tier

Figure 2-5 shows a UML class diagram for the `frontend`, `intermediate`, and `backend` packages. According to our conceptual design, the intermediate code and the symbol table are the interface between the front and back ends. For now, simply define two placeholder interfaces as framework components, `ICode` and `SymTab`, in the `intermediate` package. Also define the abstract `Backend` class as the framework component in the `backend` package.

Figure 2-5: Good symmetry of the `frontend` and `backend` packages around the `intermediate` package. The `ICode` and `SymTab` interfaces are framework components in the `intermediate` package, and the abstract `Backend` class is the framework component in the `backend` package.

The diagram shows that a framework `Parser` object in the `frontend` package and a `Backend` object in the `backend` package own intermediate code and symbol table objects. Both class `Parser` and class `Backend` have the same relationships to class `MessageHandler` and to interface `MessageProducer`. Therefore, this framework has good symmetry.

Listings 2-14 and 2-15 show interfaces `ICode` and `SymTab`, respectively. You'll fill out these interfaces in later chapters.

Listing 2-14: Interface `ICode`

```
package wci.intermediate;

public interface ICode
{
}
```

Listing 2-15: Interface `SymTab`

```
package wci.intermediate;

public interface SymTab
{
}
```

Back End

The conceptual design states that the back end will support either a compiler or an interpreter. As indicated by the class diagram in Figure 2-5, the `Backend` class in the `backend` package is also a message producer. Like the `Parser` and `Source` classes in package `frontend`, the `Backend` class in package `backend` implements the `MessageProducer` interface and delegates message handling to the `MessageHandler` helper class. The abstract `process()` method requires references to the intermediate code and to the symbol table. A compiler would implement `process()` to generate object code, and an interpreter would implement `process()` to execute the program.

Listing 2-16 shows the `process()` method of the `Backend` class. Not shown are the getter methods and the methods that implement the `MessageProducer` interface, which are done similarly to class `Parser` in the front end.

Listing 2-16: Method `process()` of the abstract `Backend` framework class

```
package wci.backend;

import wci.intermediate.ICode;
import wci.intermediate.SymTab;
import wci.message.*;

/**
 * <h1>Backend</h1>
 *
 * <p>The framework class that represents the back end component.</p>
 */
public abstract class Backend implements MessageProducer
{
    // Message handler delegate.
    protected static MessageHandler messageHandler;

    static {
        messageHandler = new MessageHandler();
    }
    protected SymTab symTab;  // symbol table
    protected ICode iCode;    // intermediate code

    /**
     * Process the intermediate code and the symbol table generated by the
     * parser.  To be implemented by a compiler or an interpreter subclass.
     * @param iCode the intermediate code.
     * @param symTab the symbol table.
     * @throws Exception if an error occurred.
     */
    public abstract void process(ICode iCode, SymTab symTab)
        throws Exception;
}
```

This completes the framework components and satisfies the first goal of this chapter. The framework components are all language independent; nowhere is there anything specific to Pascal or any other source language. The back end of the framework can support either a compiler or an interpreter.

Pascal-Specific Front End Components

Now that the framework components are in place, you're ready to define some initial Pascal-specific components. Base these latter components on the framework components. In other words, they will be subclasses of the language-independent framework classes and provide language-specific implementations of the abstract methods.

The UML diagrams in
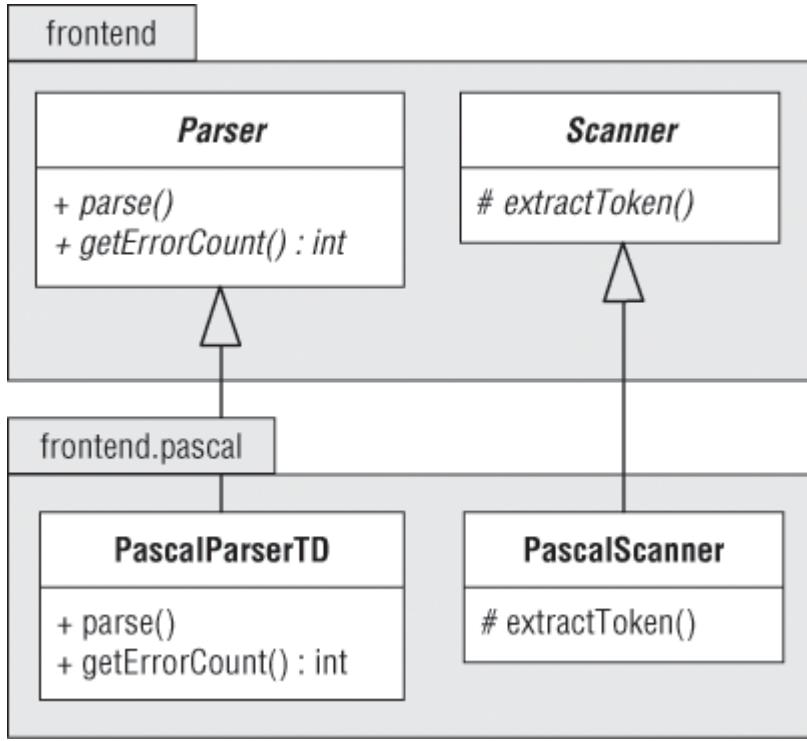
Figure 2-6 show the Pascal-specific implementations of the `Parser` and `Scanner` framework classes. Pascal-specific front end classes are defined in package `frontend.pascal`.

Pascal Parser

The initial implementation of a Pascal parser is extremely simplified. The class name `PascalParserTD` indicates the source language and the parser type. The TD stands for *top down*, which is the type of parser you'll develop in the next several chapters.[4] See Listing 2-17.

[4] The last chapter in this book briefly discusses *bottom up* parsers.

Figure 2-6: Pascal-specific implementations in the front end

Listing 2-17: The initial implementation of class `PascalParserTD`

```
package wci.frontend.pascal;

import wci.frontend.*;
import wci.message.Message;

import static wci.message.MessageType.PARSER_SUMMARY;

/**
 * <h1>PascalParserTD</h1>
 *
 * <p>The top-down Pascal parser.</p>
 */
public class PascalParserTD extends Parser
{
    /**
     * Constructor.
     * @param scanner the scanner to be used with this parser.
     */
    public PascalParserTD(Scanner scanner)
    {
        super(scanner);
    }

    /**
     * Parse a Pascal source program and generate the symbol table
     * and the intermediate code.
     */
    public void parse()
        throws Exception
    {
        Token token;
        long startTime = System.currentTimeMillis();

        while (!((token = nextToken()) instanceof EofToken)) {}

        // Send the parser summary message.
        float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
        sendMessage(new Message(PARSER_SUMMARY,
                        new Number[] {token.getLineNumber(),
                                      getErrorCount(),
                                      elapsedTime}));
    }

    /**
     * Return the number of syntax errors found by the parser.
     * @return the error count.
     */
    public int getErrorCount()
    {
        return 0;
    }
}
```

The `parse()` method implements the `Parser` superclass's abstract method. All the method does is repeatedly call the `nextToken()` method which, as you saw earlier, is defined in the superclass and calls the scanner's `nextToken()` method to construct the next token. `PascalParserTD` calls `nextToken()` repeatedly until the token type is `EofToken`; for now, it ignores other token types. Calling `nextToken()` forces the scanner to read from the source. The method also computes the total parsing time and sends a parser summary message. For now, method `getErrorCount()` simply returns 0.

By convention, the parser summary message has the format:

PARSER_SUMMARY Message	
`token.getLineNumber()`	number of source lines read

`getErrorCount()`	number of syntax errors
`elapsedTime`	elapsed parsing time

All listeners of `PARSER_SUMMARY` messages must follow this convention.

You'll develop the `parse()` method further in the following chapters.

Pascal Scanner

Our initial implementation of a Pascal scanner is also greatly simplified. The `PascalScanner` class implements the `extractToken()` method of its `Scanner` superclass. See Listing 2-18.

Listing 2-18: The initial implementation of class `PascalScanner`

```
package wci.frontend.pascal;

import wci.frontend.*;
import static wci.frontend.Source.EOF;

/**
 * <h1>PascalScanner</h1>
 *
 * <p>The Pascal scanner.</p>
 */
public class PascalScanner extends Scanner
{
    /**
     * Constructor
     * @param source the source to be used with this scanner.
     */
    public PascalScanner(Source source)
    {
        super(source);
    }

    /**
     * Extract and return the next Pascal token from the source.
     * @return the next token.
     * @throws Exception if an error occurred.
     */
    protected Token extractToken()
        throws Exception
    {
        Token token;
        char currentChar = currentChar();

        // Construct the next token.  The current character determines the
        // token type.
        if (currentChar == EOF) {
            token = new EofToken(source);
        }
        else {
            token = new Token(source);
        }

        return token;
    }
}
```

This implementation of method `extractToken()` establishes the template for how to complete the method in the next chapter. A call to the superclass's `currentChar()` method sets the current source character. As described above, the current character determines what type of token to construct. For now, since it doesn't know how to do much else, the method constructs an `EofToken` if the character is the end-of-file character `EOF`, and for any other character, it constructs a generic `Token` object.

How can you be sure that subsequent calls to `extractToken()` will each return the next token? Remember that each token's `extract()` method consumes source characters and leaves the current source line position at one beyond the last token character. Therefore, at the beginning of the next call to `extractToken()`, the call to `currentChar()` will return the first source character after the previous token.

You'll develop a complete Pascal scanner implementation in the next chapter.

Design Note

Java's access control modifiers, `public`, `protected`, and `private`, play a major role in maintaining program security, reliability, and modularity. A public field or method is accessible by any class without restrictions. A protected field or method is accessible by any class defined within the same package or by any subclasses defined in other packages. A package field or method (this is the default, since it requires no access control modifier) is accessible only by any class defined within the same package. A private field or method is accessible only within its class.

You will often use protected fields and methods to share them with other objects within the same package, such as the front end package. Protected access is especially useful when you define language-specific subclasses of the framework classes. Each subclass can then access a protected field or method of its superclass.

Public methods serve as gateways across packages. For example, the parser's `parse()` method is public so it can be called from outside the package. Limiting the number of such public methods helps to preserve modularity.

Top-level classes can themselves have public or package access to control who can reference their objects. A class defined within another class can be private.

During program design, try to specify as well as possible which access control modifiers are appropriate for each class, field, and method. A good rule of thumb is to use the most restrictive access control as practicable. However, in a program as complex as a compiler or an interpreter, it is very normal to have to go back and change existing modifiers as you develop more of the program.[5]

[5] Book authors have the advantage, before publication, of going back to earlier chapters to make such changes and thus appear in print to be mysteriously prescient and all-knowing.

A Front End Factory

The framework components in the front end are language-independent. You then integrate language-specific components into the framework. The framework can support parsers for different source languages, and even multiple types of parsers for a specific language. Also, for any specific language, the parser and the scanner are closely related.

A *factory class* makes it easier for a compiler or an interpreter to create proper front end components for specific languages. See Listing 2-19.

Listing 2-19: The `FrontendFactory` factory class

```
package wci.frontend;

import wci.frontend.pascal.PascalParserTD;
import wci.frontend.pascal.PascalScanner;

/**
 * <h1>FrontendFactory</h1>
 *
 * <p>A factory class that creates parsers for specific source languages.</p>
 */
public class FrontendFactory
{
    /**
     * Create a parser.
     * @param language the name of the source language (e.g., "Pascal").
     * @param type the type of parser (e.g., "top-down").
     * @param source the source object.
     * @return the parser.
     * @throws Exception if an error occurred.
     */
    public static Parser createParser(String language, String type,
                                      Source source)
        throws Exception
    {
        if (language.equalsIgnoreCase("Pascal") &&
            type.equalsIgnoreCase("top-down"))
        {
            Scanner scanner = new PascalScanner(source);
            return new PascalParserTD(scanner);
        }
        else if (!language.equalsIgnoreCase("Pascal")) {
            throw new Exception("Parser factory: Invalid language &apos;" +
                                language + "&apos;");
        }
        else {
            throw new Exception("Parser factory: Invalid type &apos;" +
                                type + "&apos;");
        }
    }
}
```

Static method `createParser()` does all the work. Given string arguments indicating the source language and the parser type, along with a reference to the source, it verifies the language ("`Pascal`" only for this book) and the type ("`top-down`" only). If both are proper, the method creates a `PascalScanner` object and associates it with the source. Then it creates and returns a `PascalParserTD` parser object which it associates with the scanner.

Design Note

The factory class is more than just a convenience. Because the parser and the scanner are closely related, using the factory class ensures that they're created in matched pairs. For example, a Pascal parser is always created with a Pascal scanner.

Using a factory class also preserves flexibility. The assignment

```
Parser parser = FrontendFactory.createParser( … );
```

is more flexible than

```
PascalParserTD parser = new PascalParserTD( … );
```
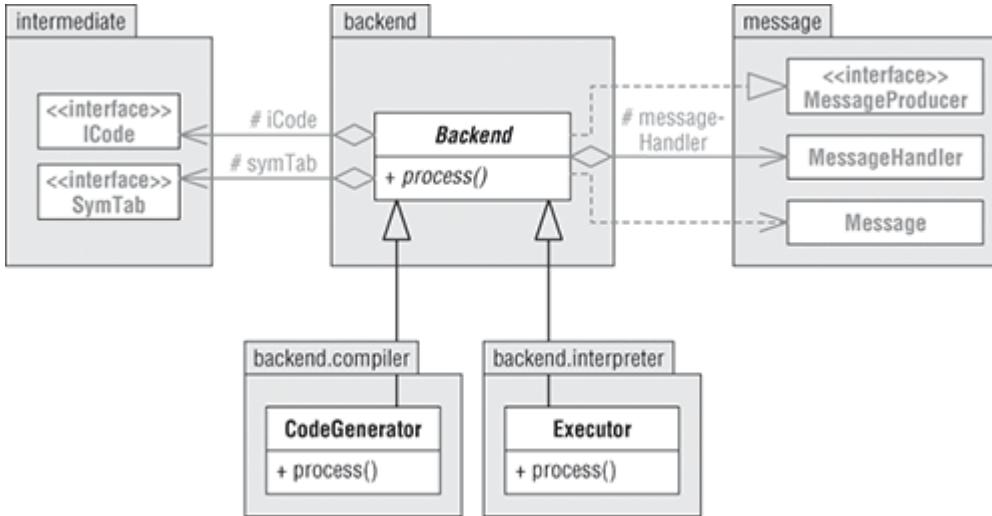
The latter permanently ties variable `parser` to a top-down Pascal parser. On the other hand, the call to the factory allows us to change the values of the arguments to create different parsers without changing any other code.

Initial Back End Implementations

The back end of the framework supports both compilers and interpreters. For now, there are two mostly stubbed-out implementations of the abstract `Backend` framework class: one for the compiler and one for the interpreter. Figure 2-7 shows the UML class diagram.

Figure 2-7: Subclasses `CodeGenerator` and `Executor` are the compiler and interpreter implementations, respectively, in the back end.

Compiler

The back end of a compiler generates object code. The `CodeGenerator` class in package `backend.compiler` implements the abstract `Backend` framework class. For now, it is mostly stubbed out. Listing 2-20 shows how it implements the `process()` method of its superclass. This method's arguments are references to the intermediate code and the symbol table, and it produces a status message that indicates how many machine-language instructions it generated (0 for now) and how much time it took to generate the object code. It calls `sendMessage()` to send a compiler summary message.

By convention, the compiler summary message has the format:

COMPILER_SUMMARY Message	
instructionCount	number of instructions generated
elapsedTime	elapsed code generation time

All listeners of `COMPILER_SUMMARY` messages must follow this convention.

Listing 2-20: The initial rudimentary `process()` method of class `CodeGenerator`

```
/**
 * Process the intermediate code and the symbol table generated by the
 * parser to generate machine-language instructions.
 * @param iCode the intermediate code.
 * @param symTab the symbol table.
 * @throws Exception if an error occurred.
 */
public void process(ICode iCode, SymTab symTab)
    throws Exception
{
    long startTime = System.currentTimeMillis();
    float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
    int instructionCount = 0;

    // Send the compiler summary message.
    sendMessage(new Message(COMPILER_SUMMARY,
                    new Number[] {instructionCount,
                                    elapsedTime}));
}
```

Interpreter

The back end of an interpreter executes the source program. The `Executor` class in package `backend.interpreter` implements the abstract `Backend` framework class. For now, it is also mostly stubbed out. Listing 2-21 shows how it implements the `process()` method of its superclass. It produces an interpreter status message that indicates how many statements it executed and the number of runtime errors (both 0 for now) and how much time it took to execute the source program. It calls `sendMessage()` to send the message.

By convention, the interpreter summary message has the format:

INTERPRETER_SUMMARY Message	
executionCount	number of statements executed
runtimeErrors	number of runtime errors
elapsedTime	elapsed execution time

All listeners of `INTERPRETER_SUMMARY` messages must follow this convention.

Listing 2-21: The initial rudimentary `process()` method of class `Executor`

```
/**
 * Process the intermediate code and the symbol table generated by the
 * parser to execute the source program.
 * @param iCode the intermediate code.
 * @param symTab the symbol table.
 * @throws Exception if an error occurred.
 */
public void process(ICode iCode, SymTab symTab)
    throws Exception
{
```

```
            long startTime = System.currentTimeMillis();
            float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
            int executionCount = 0;
            int runtimeErrors = 0;

            // Send the interpreter summary message.
            sendMessage(new Message(INTERPRETER_SUMMARY,
                                    new Number[] {executionCount,
                                                  runtimeErrors,
                                                  elapsedTime}));
    }
```

A Back End Factory

Like the one in the front end, a back end factory class creates proper back end components. See Listing 2-22.

Listing 2-22: The `BackendFactory` factory class

```
package wci.backend;

import wci.backend.compiler.CodeGenerator;
import wci.backend.interpreter.Executor;

/**
 * <h1>BackendFactory</h1>
 *
 * <p>A factory class that creates compiler and interpreter components.</p>
 */
public class BackendFactory
{
    /**
     * Create a compiler or an interpreter back end component.
     * @param operation either "compile" or "execute"
     * @return a compiler or an interpreter back end component.
     * @throws Exception if an error occurred.
     */
    public static Backend createBackend(String operation)
        throws Exception
    {
        if (operation.equalsIgnoreCase("compile")) {
            return new CodeGenerator();
        }
        else if (operation.equalsIgnoreCase("execute")) {
            return new Executor();
        }
        else {
            throw new Exception("Backend factory: Invalid operation &apos;" +
                                operation + "&apos;");
        }
    }
}
```

Static method `createBackend()` verifies a string argument that indicates whether to create a compiler back end ("`compile`") or an interpreter back end ("`execute`"). If the argument is proper, the method creates and returns the appropriate back end component.

You've now successfully satisfied the second and third goals of this chapter. You integrated initial Pascal-specific components into the front end of the framework and initial compiler and interpreter components into the back end.

Program 2: Program Listings

The framework components and the initial implementation components are all in place and integrated. Some simple end-to-end tests will verify that you've designed and developed these components correctly. A compiler test will cause the front end to parse a source Pascal program, generate a listing, and print the message produced by the compiler back end. Similarly, an interpreter test will cause the front end to parse a source Pascal program, generate a listing, and print the message produced by the interpreter back end.

Listing 2-23 shows the main `Pascal` class. Besides being the test program, it is the means by which you will either compile or execute a Pascal source program.

Listing 2-23: The main class `Pascal` which compiles or interprets a Pascal source program

```
import java.io.BufferedReader;
import java.io.FileReader;

import wci.frontend.*;
import wci.intermediate.*;
import wci.backend.*;
import wci.message.*;

import static wci.message.MessageType.*;

/**
 * <h1>Pascal</h1>
 *
 * <p>Compile or interpret a Pascal source program.</p>
 */
public class Pascal
{
    private Parser parser;    // language-independent parser
    private Source source;    // language-independent scanner
    private ICode iCode;      // generated intermediate code
    private SymTab symTab;    // generated symbol table
    private Backend backend;  // backend

    /**
     * Compile or interpret a Pascal source program.
     * @param operation either "compile" or "execute".
     * @param filePath the source file path.
     * @param flags the command line flags.
     */
    public Pascal(String operation, String filePath, String flags)
    {
```

```java
            try {
                boolean intermediate = flags.indexOf('i') > -1;
                boolean xref         = flags.indexOf('x') > -1;

                source = new Source(new BufferedReader(new FileReader(filePath)));
                source.addMessageListener(new SourceMessageListener());

                parser = FrontendFactory.createParser("Pascal", "top-down", source);
                parser.addMessageListener(new ParserMessageListener());

                backend = BackendFactory.createBackend(operation);
                backend.addMessageListener(new BackendMessageListener());

                parser.parse();
                source.close();

                iCode = parser.getICode();
                symTab = parser.getSymTab();

                backend.process(iCode, symTab);
            }
            catch (Exception ex) {
                System.out.println("***** Internal translator error. *****");
                ex.printStackTrace();
            }
    }

    private static final String FLAGS = "[-ix]";
    private static final String USAGE =
        "Usage: Pascal execute|compile " + FLAGS + " <source file path>";

    /**
     * The main method.
     * @param args command-line arguments: "compile" or "execute" followed by
     *             optional flags followed by the source file path.
     */
    public static void main(String args[])
    {
        try {
            String operation = args[0];

            // Operation.
            if (!(   operation.equalsIgnoreCase("compile")
                  || operation.equalsIgnoreCase("execute"))) {
                throw new Exception();
            }

            int i = 0;
            String flags = "";

            // Flags.
            while ((++i < args.length) && (args[i].charAt(0) == '-')) {
                flags += args[i].substring(1);
            }

            // Source path.
            if (i < args.length) {
                String path = args[i];
                new Pascal(operation, path, flags);
            }
            else {
                throw new Exception();
            }
        }
        catch (Exception ex) {
            System.out.println(USAGE);
        }
    }

    private static final String SOURCE_LINE_FORMAT = "%03d %s";

    /**
     * Listener for source messages.
     */
    private class SourceMessageListener implements MessageListener
    {
        /**
         * Called by the source whenever it produces a message.
         * @param message the message.
         */
        public void messageReceived(Message message)
        {
            MessageType type = message.getType();
            Object body[] = (Object []) message.getBody();

            switch (type) {

                case SOURCE_LINE: {
                    int lineNumber = (Integer) body[0];
                    String lineText = (String) body[1];

                    System.out.println(String.format(SOURCE_LINE_FORMAT,
                                                     lineNumber, lineText));
                    break;
                }
            }
        }
    }

    private static final String PARSER_SUMMARY_FORMAT =
        "\n%,20d source lines." +
        "\n%,20d syntax errors." +
        "\n%,20.2f seconds total parsing time.\n";

    /**
     * Listener for parser messages.
     */
    private class ParserMessageListener implements MessageListener
    {
        /**
         * Called by the parser whenever it produces a message.
         * @param message the message.
         */
        public void messageReceived(Message message)
        {
            MessageType type = message.getType();

            switch (type) {

                case PARSER_SUMMARY: {
                    Number body[] = (Number[]) message.getBody();
                    int statementCount = (Integer) body[0];
```

```java
                        int syntaxErrors = (Integer) body[1];
                        float elapsedTime = (Float) body[2];

                        System.out.printf(PARSER_SUMMARY_FORMAT,
                                          statementCount, syntaxErrors,
                                          elapsedTime);
                        break;
                    }
                }
            }
        }
    }

    private static final String INTERPRETER_SUMMARY_FORMAT =
        "\n%,20d statements executed." +
        "\n%,20d runtime errors." +
        "\n%,20.2f seconds total execution time.\n";

    private static final String COMPILER_SUMMARY_FORMAT =
        "\n%,20d instructions generated." +
        "\n%,20.2f seconds total code generation time.\n";

    /**
     * Listener for back end messages.
     */
    private class BackendMessageListener implements MessageListener
    {
        /**
         * Called by the back end whenever it produces a message.
         * @param message the message.
         */
        public void messageReceived(Message message)
        {
            MessageType type = message.getType();

            switch (type) {

                case INTERPRETER_SUMMARY: {
                    Number body[] = (Number[]) message.getBody();
                    int executionCount = (Integer) body[0];
                    int runtimeErrors = (Integer) body[1];
                    float elapsedTime = (Float) body[2];

                    System.out.printf(INTERPRETER_SUMMARY_FORMAT,
                                      executionCount, runtimeErrors,
                                      elapsedTime);
                    break;
                }

                case COMPILER_SUMMARY: {
                    Number body[] = (Number[]) message.getBody();
                    int instructionCount = (Integer) body[0];
                    float elapsedTime = (Float) body[1];

                    System.out.printf(COMPILER_SUMMARY_FORMAT,
                                      instructionCount, elapsedTime);
                    break;
                }
            }
        }
    }
}
```

If the `classes` directory contains the class files and the current directory contains the Pascal source file `hello.pas`, the command line to compile the source file would be similar to:

```
java –classpath classes Pascal compile hello.pas
```

The command line to interpret the source file would be similar to:

```
java –classpath classes Pascal execute hello.pas
```

The `main()` method verifies the command line's back end type and source file path arguments and creates a new `Pascal` object, passing the type and path to the constructor. The optional `-i` (intermediate) and `-x` (cross-reference) arguments will be useful in later chapters to control the amount of output.

The constructor creates a new `Source` object from the source file path. It uses the front end and back end factories to create proper components based on the command-line argument values. The constructor calls the parser's `parse()` method to parse the source file. It gets the generated intermediate code and the symbol table from the parser and passes them to the back end's `process()` method.

Note that the constructor calls the front end factory to create a top-down parser for the Pascal source language but otherwise doesn't depend on what the source language is or what type of parser is used. Indeed, the source language and the parser type could have been passed in on the command line. Likewise, the class doesn't depend on whether the back end factory created components for a compiler or for an interpreter.

The three inner classes are listeners for parser, source, and back end messages. The `messageReceived()` methods of classes `SourceMessageListener`, `ParserMessageListener`, and `BackendMessageListener` follow the message format conventions of the source, parser, and the back end, respectively. They use the format strings `SOURCE_LINE_FORMAT`, `PARSER_SUMMARY_FORMAT`, `INTERPRETER_SUMMARY_FORMAT`, and `COMPILER_SUMMARY_FORMAT` to output appropriately formatted text to `System.out`.

Assume the file `hello.pas` contains the Pascal program in Listing 2-24.

Listing 2-24: Pascal program `hello.pas`

```pascal
PROGRAM hello (output);

{Write &apos;Hello, world.&apos; ten times.}

VAR
    i : integer;

BEGIN {hello}
    FOR i := 1 TO 10 DO BEGIN
        writeln(&apos;Hello, world.&apos;);
    END;
END {hello}.
```

The rudimentary compiler will generate the output shown in Listing 2-25.

Listing 2-25: Pascal "compiler" output

```
001 PROGRAM hello (output);
002
003 {Write &apos;Hello, world.&apos; ten times.}
004
005 VAR
006     i : integer;
007
008 BEGIN {hello}
009     FOR i := 1 TO 10 DO BEGIN
010         writeln(&apos;Hello, world.&apos;);
011     END;
012 END {hello}.

                    12 source statements.
                     0 syntax errors.
                  0.01 seconds total parsing time.

                     0 instructions generated.
                  0.00 seconds total code generation time.
```

Design Note

That was certainly a lot of trouble in this chapter to ensure that the framework is source language-independent and can support either a compiler or an interpreter. Even the `Pascal` class was fairly agnostic. Then you added Pascal-specific components to the front end, and compiler- and interpreter-specific components to the back end.

One of the major software engineering challenges is managing change. You may change the source language or the type of parser, or whether you want a compiler or an interpreter. Because the framework is designed to be flexible, it should be solid and not require any modifications. Instead, you accommodated the changes with the implementation subclasses.

If, say, you want to use a bottom-up Pascal parser instead of top-down, add a new `Parser` implementation subclass `PascalParserBU` which would use the same `PascalScanner` class. Or, change the source language to Java and add the `Parser` subclass `JavaParserTD`, which will require a new `Scanner` subclass `JavaScanner`.

Therefore, to manage change, apply the software engineering principle that says to encapsulate the code that will vary in order to isolate it from the code that won't. In the front end, the parser and scanner can vary, so encapsulate them in the implementation subclasses. The back end compiler and interpreter are also subclasses. These varying subclasses are isolated from the unvarying framework classes.

Because all our parsers, of which `PascalParserTD` is one example, are derived from the `Parser` base class, they constitute an interchangeable family of classes. Other families are the subclasses of the `Scanner` base class (such as `PascalScanner`), the subclasses of the `Token` base class (such as `EofToken`), and the subclasses of the `Backend` base class (`CodeGenerator` and `Executor`). Within each family, you should be able to add and swap family members without requiring code changes in the framework.

The Strategy Design Pattern specifies such use of class families. Implementing this design pattern early to manage change will return great dividends as you do further development in the following chapters.

The rudimentary interpreter will generate the output shown in Listing 2-26.

Listing 2-26: Pascal "interpreter" output

```
001 PROGRAM hello (output);
002
003 {Write &apos;Hello, world.&apos; ten times.}
004
005 VAR
006     i : integer;
007
008 BEGIN {hello}
009     FOR i := 1 TO 10 DO BEGIN
010         writeln(&apos;Hello, world.&apos;);
011     END;
012 END {hello}.

                    12 source statements.
                     0 syntax errors.
                  0.01 seconds total parsing time.

                     0 statements executed.
                     0 runtime errors.
                  0.00 seconds total execution time.
```

These end-to-end test runs satisfy the fourth and final goal of this chapter.

Chapter 3

Scanning

The scanner is the component in the front end of a compiler or an interpreter that performs the syntactic actions of reading the source program and breaking it apart into tokens. The parser calls the scanner each time it wants the next token from the source program. In this chapter, you'll design and develop a scanner for Pascal tokens.

Goals and Approach
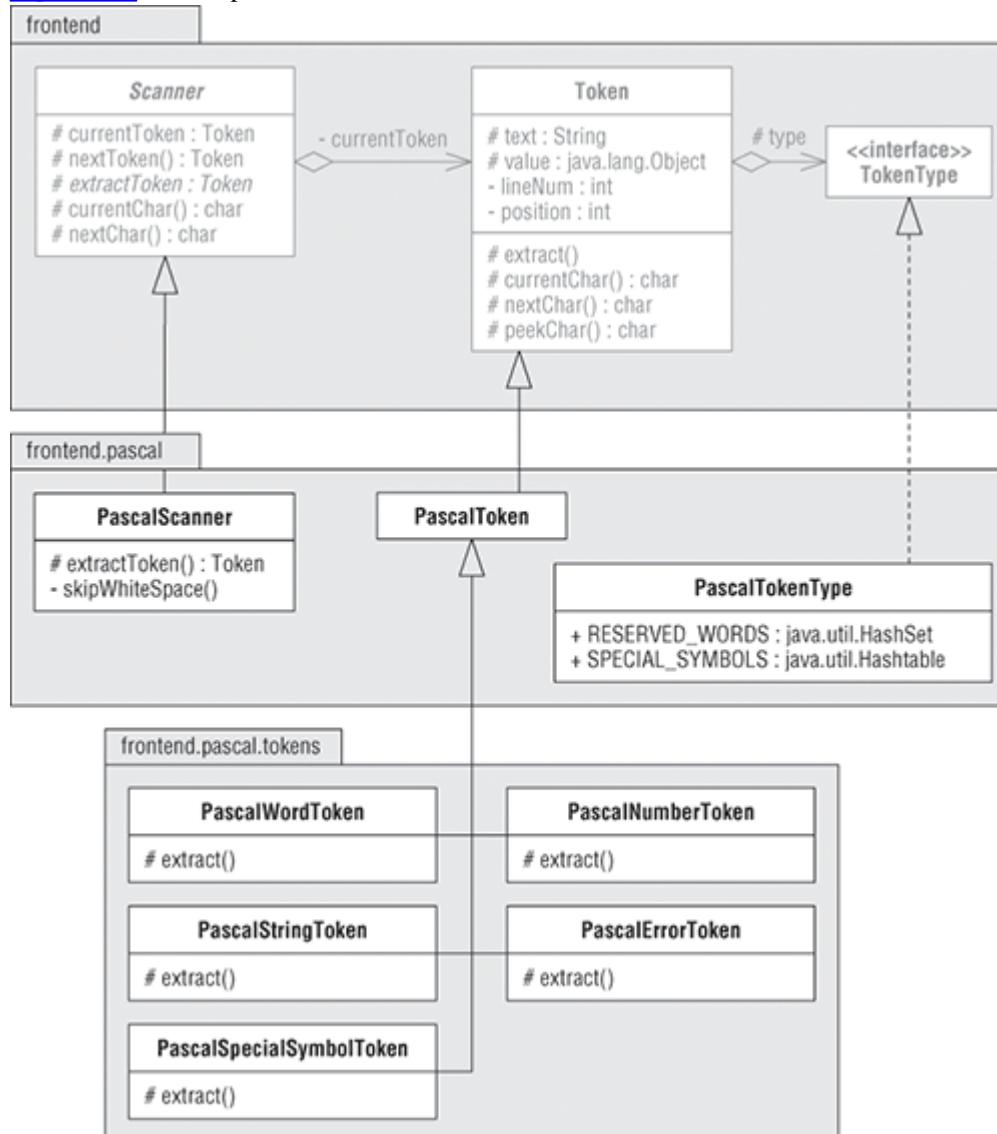
In the previous chapter, you implemented a language-independent `Scanner` framework class in the front end and a rudimentary Pascal-specific `PascalScanner` subclass of the framework class. The goals of this chapter are to complete the design and development of the Pascal scanner and then test it. The scanner will be able to:

- Extract Pascal word, number, string, and special symbol tokens from the source program.
- Determine whether a word token is an identifier or a Pascal reserved word.
- Calculate the value of a number token and determine whether its type is integer or real.
- Perform syntax error handing.

This chapter's approach is to develop Pascal-specific token subclasses that `PascalScanner` can construct. You'll create a `PascalToken` subclass of the framework `Token` class and then develop subclasses of `PascalToken` to represent Pascal's word, number, string, and special symbol tokens. You'll define Pascal token types that implement the `TokenType` interface of the previous chapter. To exercise and test the scanner, a "tokenizer" utility will read a Pascal source program and extract and list all the tokens.

Figure 3-1 summarizes our scanner design.

Figure 3-1: Pascal-specific scanner and token classes

[Figure 3-1](#) shows in gray the language-independent framework classes you've already created in the `frontend` package. You will now complete the Pascal-specific subclasses in the `frontend.pascal` package. You'll also create the new `frontend.pascal.tokens` package to contain the subclasses of `PascalToken`.

Design Note

Defining separate subclasses for the individual types of Pascal token is another use of the Strategy Design Pattern. Here, the strategies are the algorithms for constructing the different types of tokens from the source. Encapsulating each strategy (implemented by the `extract()` method) in its own subclass makes the code more modular and makes it easier for the scanner to select the appropriate type of token to construct. All the token subclasses will extend the `PascalToken` class, which will in turn extend the language-independent `Token` class.

[Figure 3-1](#) also illustrates the design principle mentioned at the beginning of the previous chapter: Always build on code that already works. Here, you're adding subclasses to the working framework that you had developed earlier.

As mentioned in Chapter 1, a scanner is also called a lexical analyzer, and its scanning operation is also called lexical analysis. Tokens are also known as *lexemes*.

Program 3: Pascal Tokenizer

Begin with the parser subclass `PascalParserTD` in package `frontend.pascal`, which you first developed in the previous chapter. Expand its `parse()` method to exercise and test the completed Pascal scanner. While you have not yet written the various token subclasses that it needs, the method shows how to scan for the various types of Pascal tokens and how to handle errors. See [Listing 3-1](#).

Listing 3-1: Methods `parse()` and `getErrorCount()` of class `PascalParserTD`

```
protected static PascalErrorHandler errorHandler = new PascalErrorHandler();

/**
 * Parse a Pascal source program and generate the symbol table
 * and the intermediate code.
 */
public void parse()
    throws Exception
{
    Token token;
    long startTime = System.currentTimeMillis();

    try {
        // Loop over each token until the end of file.
        while (!((token = nextToken()) instanceof EofToken)) {
            TokenType tokenType = token.getType();

            if (tokenType != ERROR) {

                // Format each token.
                sendMessage(new Message(TOKEN,
                                    new Object[] {token.getLineNumber(),
                                                  token.getPosition(),
                                                  tokenType,
                                                  token.getText(),
                                                  token.getValue()}));
            }
            else {
                errorHandler.flag(token, (PascalErrorCode) token.getValue(),
                                  this);
            }
        }

        // Send the parser summary message.
        float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
        sendMessage(new Message(PARSER_SUMMARY,
                            new Number[] {token.getLineNumber(),
                                          getErrorCount(),
                                          elapsedTime}));
    }
    catch (java.io.IOException ex) {
        errorHandler.abortTranslation(IO_ERROR, this);
    }
}

/**
 * Return the number of syntax errors found by the parser.
 * @return the error count.
 */
public int getErrorCount()
{
    return errorHandler.getErrorCount();
}
```

Method `parse()` loops over each token that it successively obtains from the scanner by calling `getToken()`. For each token, it sends a `TOKEN` message to all listeners indicating the token's type, text string, and source line number and starting position. The message includes the value of integer, real, and string tokens. However, if there is an error, the method delegates error handling to an instance of class `PascalErrorHandler`.

By convention, the format of the `TOKEN` message is:

TOKEN Message	
`token.getLineNumber()`	source line number
`token.getPosition()`	beginning source position
`token.getType()`	token type
`token.getText()`	token text
`token.getValue()`	token value

[Listing 3-1](#) also shows the new version of method `getErrorCount()`, which now returns the count of syntax errors that it gets from the error handler.

The private inner class `ParserMessageListener` in the main `Pascal` class listens to messages from the parser. In the previous chapter, it could process `PARSER_SUMMARY` messages. Now add the capability to process `TOKEN` and `SYNTAX_ERROR` messages from the parser. See [Listing 3-2](#).

[Listing 3-2:](#) Processing `TOKEN` and `SYNTAX_ERROR` messages by inner class `ParserMessageListener` in the main `Pascal` class

```
private static final String TOKEN_FORMAT =
    ">>> %-15s line=%03d, pos=%2d, text=\"%s\"";
private static final String VALUE_FORMAT =
    ">>>               value=%s";

private static final int PREFIX_WIDTH = 5;

/**
 * Listener for parser messages.
 */
private class ParserMessageListener implements MessageListener
{
    /**
     * Called by the parser whenever it produces a message.
     * @param message the message.
     */
    public void messageReceived(Message message)
    {
        MessageType type = message.getType();

        switch (type) {

            case TOKEN: {
                Object body[] = (Object []) message.getBody();
                int line = (Integer) body[0];
                int position = (Integer) body[1];
                TokenType tokenType = (TokenType) body[2];
                String tokenText = (String) body[3];
                Object tokenValue = body[4];

                System.out.println(String.format(TOKEN_FORMAT,
                                                 tokenType,
                                                 line,
                                                 position,
                                                 tokenText));
                if (tokenValue != null) {
                    if (tokenType == STRING) {
                        tokenValue = "\"" + tokenValue + "\"";
                    }

                    System.out.println(String.format(VALUE_FORMAT,
                                                     tokenValue));
                }

                break;
            }

            case SYNTAX_ERROR: {
                Object body[] = (Object []) message.getBody();
                int lineNumber = (Integer) body[0];
                int position = (Integer) body[1];
                String tokenText = (String) body[2];
                String errorMessage = (String) body[3];

                int spaceCount = PREFIX_WIDTH + position;
                StringBuilder flagBuffer = new StringBuilder();

                // Spaces up to the error position.
                for (int i = 1; i < spaceCount; ++i) {
                    flagBuffer.append(' ');
                }

                // A pointer to the error followed by the error message.
                flagBuffer.append("^\n*** ").append(errorMessage);

                // Text, if any, of the bad token.
                if (tokenText != null) {
                    flagBuffer.append(" [at \"").append(tokenText)
                        .append("\"]");
                }

                System.out.println(flagBuffer.toString());
                break;
            }
        }
    }
}
```

The `messageReceived()` method follows the format conventions of both the `TOKEN` messages generated by class `PascalParserTD` and the `SYNTAX_ERROR` messages generated by class `PascalErrorHandler`, which is described below. For `SYNTAX_ERROR` messages, the method displays a ^ character that points to the first character of the erroneous token and an error message.

If the `classes` directory contains the class files and current directory contains the source file `scannertest.txt`, the command line to run this test program would be similar to:

```
java -classpath classes Pascal compile scannertest.txt
```

[Listing 3-3](#) shows the sample source file `scannertest.txt` containing various Pascal tokens, some of which have errors.

[Listing 3-3:](#) Sample source file `scannertest.txt` containing various Pascal tokens, some of which have errors

```
{This is a comment.}

{This is a comment
 that spans several
 source lines.}

Two{comments in}{a row} here

{Word tokens}
Hello world
begin BEGIN Begin BeGiN begins
```

```
{String tokens}
&apos;Hello, world.&apos;
&apos;It&apos;&apos;s Friday!&apos;
&apos;&apos;
&apos; &apos;&apos; &apos;&apos; &apos;    &apos;&apos;&apos;&apos;&apos;&apos;
&apos;This string
spans
source lines.&apos;

{Special symbol tokens}
+ - * / := . , ; : = <> < <= >= > ( ) [ ] { } } ^ ..
+-:=<>=<==.....

{Number tokens}
0 1 20 00000000000000000032  31415926
3.1415926  3.14159265358979323384626433
0.00031415926E4  0.00031415926e+00004  31415.926e-4
31415926000000000000000000000e-30

{Decimal point or ..}
3.14  3..14

{Bad tokens}
123e99  123456789012345  1234.56E.  3.  5..  .14  314.e-2
What?
&apos;String not closed
```

[Listing 3-4](#) shows the output from the Pascal tokenizer utility. A three-digit line number precedes each source line, and following each source line that contains tokens, lines that start with >>> describe the tokens. Each word token is either an IDENTIFIER or a reserved word (indicated by the reserved word itself), and each INTEGER and REAL number token has a value. Each STRING token also has a value.

Listing 3-4: Output from the Pascal tokenizer

```
001 {This is a comment.}
002
003 {This is a comment
004  that spans several
005  source lines.}
006
007 Two{comments in}{a row} here
>>> IDENTIFIER      line=007, pos= 0, text="Two"
>>> IDENTIFIER      line=007, pos=24, text="here"
008
009 {Word tokens}
010 Hello world
>>> IDENTIFIER      line=010, pos= 0, text="Hello"
>>> IDENTIFIER      line=010, pos= 6, text="world"
011 begin BEGIN Begin BeGiN begins
>>> BEGIN           line=011, pos= 0, text="begin"
>>> BEGIN           line=011, pos= 6, text="BEGIN"
>>> BEGIN           line=011, pos=12, text="Begin"
>>> BEGIN           line=011, pos=18, text="BeGiN"
>>> IDENTIFIER      line=011, pos=24, text="begins"
012
013 {String tokens}
014 &apos;Hello, world.&apos;
>>> STRING          line=014, pos= 0, text="&apos;Hello, world.&apos;"
>>>                 value="Hello, world."
015 &apos;It&apos;&apos;s Friday!&apos;
>>> STRING          line=015, pos= 0, text="&apos;It&apos;&apos;s Friday!&apos;"
>>>                 value="It&apos;s Friday!"
016 &apos;&apos;
>>> STRING          line=016, pos= 0, text="&apos;&apos;"
>>>                 value=""
017 &apos; &apos;&apos; &apos;&apos; &apos;    &apos;&apos;&apos;&apos;&apos;&apos;
>>> STRING          line=017, pos= 0, text="&apos; &apos;&apos; &apos;&apos; &apos;"
>>>                 value=" &apos; &apos; "
>>> STRING          line=017, pos=12, text="&apos;&apos;&apos;&apos;&apos;&apos;"
>>>                 value="&apos;&apos;"
018 &apos;This string
019 spans
020 source lines.&apos;
>>> STRING          line=018, pos= 0, text="&apos;This string spans source lines.&apos;"
>>>                 value="This string spans source lines."
021
022 {Special symbol tokens}
023 + - * / := . , ; : = <> < <= >= > ( ) [ ] { } } ^ ..
>>> PLUS            line=023, pos= 0, text="+"
>>> MINUS           line=023, pos= 2, text="-"
>>> STAR            line=023, pos= 4, text="*"
>>> SLASH           line=023, pos= 6, text="/"
>>> COLON_EQUALS    line=023, pos= 8, text=":="
>>> DOT             line=023, pos=11, text="."
>>> COMMA           line=023, pos=13, text=","
>>> SEMICOLON       line=023, pos=15, text=";"
>>> COLON           line=023, pos=17, text=":"
>>> EQUALS          line=023, pos=19, text="="
>>> NOT_EQUALS      line=023, pos=21, text="<>"
>>> LESS_THAN       line=023, pos=24, text="<"
>>> LESS_EQUALS     line=023, pos=26, text="<="
>>> GREATER_EQUALS  line=023, pos=29, text=">="
>>> GREATER_THAN    line=023, pos=32, text=">"
>>> LEFT_PAREN      line=023, pos=34, text="("
>>> RIGHT_PAREN     line=023, pos=36, text=")"
>>> LEFT_BRACKET    line=023, pos=38, text="["
>>> RIGHT_BRACKET   line=023, pos=40, text="]"
>>> RIGHT_BRACE     line=023, pos=46, text="}"
>>> UP_ARROW        line=023, pos=48, text="^"
>>> DOT_DOT         line=023, pos=50, text=".."
024 +-:=<>=<==.....
>>> PLUS            line=024, pos= 0, text="+"
>>> MINUS           line=024, pos= 1, text="-"
>>> COLON_EQUALS    line=024, pos= 2, text=":="
>>> NOT_EQUALS      line=024, pos= 4, text="<>"
>>> EQUALS          line=024, pos= 6, text="="
>>> LESS_EQUALS     line=024, pos= 7, text="<="
>>> EQUALS          line=024, pos= 9, text="="
>>> DOT_DOT         line=024, pos=10, text=".."
>>> DOT_DOT         line=024, pos=12, text=".."
>>> DOT             line=024, pos=14, text="."
025
026 {Number tokens}
027 0 1 20 00000000000000000032  31415926
>>> INTEGER         line=027, pos= 0, text="0"
```

```
>>>                    value=0
>>> INTEGER          line=027, pos= 2, text="1"
>>>                    value=1
>>> INTEGER          line=027, pos= 4, text="20"
>>>                    value=20
>>> INTEGER          line=027, pos= 7, text="00000000000000000032"
>>>                    value=32
>>> INTEGER          line=027, pos=29, text="31415926"
>>>                    value=31415926
028 3.1415926  3.14159265358979732384626433
>>> REAL             line=028, pos= 0, text="3.1415926"
>>>                    value=3.1415925
>>> REAL             line=028, pos=11, text="3.14159265358979732384626433"
>>>                    value=3.1415927
029 0.00031415926E4  0.00031415926e+00004  31415.926e-4
>>> REAL             line=029, pos= 0, text="0.00031415926E4"
>>>                    value=3.1415925
>>> REAL             line=029, pos=17, text="0.00031415926e+00004"
>>>                    value=3.1415925
>>> REAL             line=029, pos=39, text="31415.926e-4"
>>>                    value=3.1415925
030 314159260000000000000000000000e-30
>>> REAL             line=030, pos= 0, text="314159260000000000000000000000e-30"
>>>                    value=3.1415925
031
032 {Decimal point or ..}
033 3.14  3..14
>>> REAL             line=033, pos= 0, text="3.14"
>>>                    value=3.14
>>> INTEGER          line=033, pos= 6, text="3"
>>>                    value=3
>>> DOT_DOT          line=033, pos= 7, text=".."
>>> INTEGER          line=033, pos= 9, text="14"
>>>                    value=14
034
035 {Bad tokens}
036 123e99  123456789012345  1234.56E.  3.  5..  .14  314.e-2
         ^
*** Real literal out of range [at "123e99"]
                  ^
*** Integer literal out of range [at "123456789012345"]
                                 ^
*** Invalid number [at "1234.56E"]
>>> DOT              line=036, pos=33, text="."
                                    ^
*** Invalid number [at "3."]
>>> INTEGER          line=036, pos=40, text="5"
>>>                    value=5
>>> DOT_DOT          line=036, pos=41, text=".."
>>> DOT              line=036, pos=45, text="."
>>> INTEGER          line=036, pos=46, text="14"
>>>                    value=14
                                           ^
*** Invalid number [at "314."]
>>> IDENTIFIER       line=036, pos=54, text="e"
>>> MINUS            line=036, pos=55, text="-"
>>> INTEGER          line=036, pos=56, text="2"
>>>                    value=2
037 What?
>>> IDENTIFIER       line=037, pos= 0, text="What"
         ^
*** Invalid character [at "?"]
038 &apos;String not closed
    ^
*** Unexpected end of file [at "&apos;String not closed "]

                   38 source statements.
                    7 syntax errors.
                 0.09 seconds total parsing time.

                    0 instructions generated.
                 0.00 seconds total code generation time.
```

The Pascal scanner will need to handle some challenging cases. Examples in Listing 3-4 include source line 011 (Pascal words are case insensitive), lines 015, 016, and 017 (pairs of adjacent single-quotes), and line 033 (distinguishing between a . decimal point token and the .. token between numbers). Lines 036, 037, and 038 show some simple error handling. It will flag each syntax error with a ^ that points to the first character of the erroneous token and an error message.

Syntax Error Handling

A parser must be able to handle syntax errors in the source program. Error handling is a three-step process:

    1. *Detection.* Detect the presence of a syntax error.

    2. *Flagging.* Flag the error by pointing it out or highlighting it, and display a descriptive error message.

    3. *Recovery.* Move past the error and resume parsing.

For now, error recovery will be simple: Just move on and, starting with the current character, attempt to construct the next token.[1]

---

[1] Of course, the simplest possible error recovery is for the parser to quit after encountering the first syntax error. But you can do better than that.

Listing 3-1 showed that class `PascalParserTD` delegates error handling to class `PascalErrorHandler`. Listing 3-5 shows the key methods of this delegate class. The getter method for field `errorCount` is not shown.

Listing 3-5: Key methods of class `PascalErrorHandler`

```
package wci.frontend.pascal;

import wci.frontend.*;
import wci.message.Message;

import static wci.frontend.pascal.PascalTokenType.*;
import static wci.frontend.pascal.PascalErrorCode.*;
import static wci.message.MessageType.SYNTAX_ERROR;

/**
 * <h1>PascalErrorHandler</h1>
```

```
 *
 * <p>Error handler Pascal syntax errors.</p>
 */
public class PascalErrorHandler
{
    private static final int MAX_ERRORS = 25;

    private static int errorCount = 0;   // count of syntax errors

    /**
     * Flag an error in the source line.
     * @param token the bad token.
     * @param errorCode the error code.
     * @param parser the parser.
     * @return the flagger string.
     */
    public void flag(Token token, PascalErrorCode errorCode, Parser parser)
    {
        // Notify the parser&apos;s listeners.
        parser.sendMessage(new Message(SYNTAX_ERROR,
                                new Object[] {token.getLineNumber(),
                                              token.getPosition(),
                                              token.getText(),
                                              errorCode.toString()}));

        if (++errorCount > MAX_ERRORS) {
            abortTranslation(TOO_MANY_ERRORS, parser);
        }
    }

    /**
     * Abort the translation.
     * @param errorCode the error code.
     * @param parser the parser.
     */
    public void abortTranslation(PascalErrorCode errorCode, Parser parser)
    {
        // Notify the parser&apos;s listeners and then abort.
        String fatalText = "FATAL ERROR: " + errorCode.toString();
        parser.sendMessage(new Message(SYNTAX_ERROR,
                                new Object[] {0,
                                              0,
                                              "",
                                              fatalText}));
        System.exit(errorCode.getStatus());
    }
}
```

Method `flag()` constructs a message that it sends to the parser's listeners. By convention, the format of this SYNTAX_ERROR message is:

SYNTAX_ERROR Message	
token.getLineNumber()	source line number
token.getPosition()	beginning source position
token.getText()	token text
errorCode.toString()	syntax error message

Method `abortTranslation()` will be called for fatal errors. It passes an exit status to `System.exit()`, which terminates the program.

Design Note

By delegating error handling to class `PascalErrorHandler`, parsing and error handling are loosely coupled. This will allow you to change in the future how to handle syntax errors without affecting the parser code. Delegation helps to limit the responsibilities of each class. The parser classes need to concern themselves only with parsing, and the error handler class only with handling errors.

Listing 3-6 shows the `PascalErrorCode` enumerated type. Each enumerated value represents an error code, and each error code has a message. For example, the error code RANGE_INTEGER has the message `"Integer literal out of range."` Error codes that represent fatal errors also have a nonzero exit status. For example, the error code IO_ERROR has the exit status -101.

Listing 3-6: Enumerated type `PascalErrorCode`

```
package wci.frontend.pascal;

/**
 * <h1>PascalErrorCode</h1>
 *
 * <p>Pascal translation error codes.</p>
 */
public enum PascalErrorCode
{
    ALREADY_FORWARDED("Already specified in FORWARD"),
    IDENTIFIER_REDEFINED("Redefined identifier"),
    IDENTIFIER_UNDEFINED("Undefined identifier"),
    INCOMPATIBLE_ASSIGNMENT("Incompatible assignment"),
    INCOMPATIBLE_TYPES("Incompatible types"),
    INVALID_ASSIGNMENT("Invalid assignment statement"),
    INVALID_CHARACTER("Invalid character"),
    INVALID_CONSTANT("Invalid constant"),
    INVALID_EXPONENT("Invalid exponent"),
    INVALID_EXPRESSION("Invalid expression"),
    INVALID_FIELD("Invalid field"),
    INVALID_FRACTION("Invalid fraction"),
    INVALID_IDENTIFIER_USAGE("Invalid identifier usage"),
    INVALID_INDEX_TYPE("Invalid index type"),
    INVALID_NUMBER("Invalid number"),
    INVALID_STATEMENT("Invalid statement"),
    INVALID_SUBRANGE_TYPE("Invalid subrange type"),
    INVALID_TARGET("Invalid assignment target"),
    INVALID_TYPE("Invalid type"),
    INVALID_VAR_PARM("Invalid VAR parameter"),
    MIN_GT_MAX("Min limit greater than max limit"),
    MISSING_BEGIN("Missing BEGIN"),
```

```
        MISSING_COLON("Missing :"),
        MISSING_COLON_EQUALS("Missing :="),
        MISSING_COMMA("Missing ,"),
        MISSING_CONSTANT("Missing constant"),
        MISSING_DO("Missing DO"),
        MISSING_DOT_DOT("Missing .."),
        MISSING_END("Missing END"),
        MISSING_EQUALS("Missing ="),
        MISSING_FOR_CONTROL("Invalid FOR control variable"),
        MISSING_IDENTIFIER("Missing identifier"),
        MISSING_LEFT_BRACKET("Missing ["),
        MISSING_OF("Missing OF"),
        MISSING_PERIOD("Missing ."),
        MISSING_PROGRAM("Missing PROGRAM"),
        MISSING_RIGHT_BRACKET("Missing ]"),
        MISSING_RIGHT_PAREN("Missing )"),
        MISSING_SEMICOLON("Missing ;"),
        MISSING_THEN("Missing THEN"),
        MISSING_TO_DOWNTO("Missing TO or DOWNTO"),
        MISSING_UNTIL("Missing UNTIL"),
        MISSING_VARIABLE("Missing variable"),
        CASE_CONSTANT_REUSED("CASE constant reused"),
        NOT_CONSTANT_IDENTIFIER("Not a constant identifier"),
        NOT_RECORD_VARIABLE("Not a record variable"),
        NOT_TYPE_IDENTIFIER ("Not a type identifier"),
        RANGE_INTEGER("Integer literal out of range"),
        RANGE_REAL("Real literal out of range"),
        STACK_OVERFLOW("Stack overflow"),
        TOO_MANY_LEVELS("Nesting level too deep"),
        TOO_MANY_SUBSCRIPTS("Too many subscripts"),
        UNEXPECTED_EOF("Unexpected end of file"),
        UNEXPECTED_TOKEN("Unexpected token"),
        UNIMPLEMENTED("Unimplemented feature"),
        UNRECOGNIZABLE("Unrecognizable input"),
        WRONG_NUMBER_OF_PARMS("Wrong number of actual parameters"),

        // Fatal errors.
        IO_ERROR(-101, "Object I/O error"),
        TOO_MANY_ERRORS(-102, "Too many syntax errors");

        private int status;     // exit status
        private String message; // error message

        /**
         * Constructor.
         * @param message the error message.
         */
        PascalErrorCode(String message)
        {
            this.status = 0;
            this.message = message;
        }

        /**
         * Constructor.
         * @param status the exit status
         * @param message the error message.
         */
        PascalErrorCode(int status, String message)
        {
            this.status = status;
            this.message = message;
        }

        /**
         * Getter.
         * @return the exit status.
         */
        public int getStatus()
        {
            return status;
        }

        /**
         * @return the message.
         */   public String toString()
        {
            return message;
        }
    }
}
```

How does the parser know that an error occurred? Adopt the following rules:

- If an instance of one of the Pascal token subclasses (such as `PascalNumberToken`) finds a syntax error, it will set its type field to the `PascalTokenType` enumerated value `ERROR` and its value field to the appropriate `PascalErrorCode` enumerated value.
- If the scanner finds a syntax error (such as an invalid character that cannot start a legitimate Pascal token), it will construct a `PascalErrorToken`.

If the scanner and token classes follow these rules, the parser can check for errors, as you saw in <u>Listing 3-1</u>:

```
            TokenType tokenType = token.getType();
            if (tokenType != ERROR) ...
```

Exception handlers can catch fatal errors. <u>Listing 3-1</u> also showed catching an I/O error and then aborting the translation:

```
        catch (java.io.IOException ex) {
            errorHandler.abortTranslation(IO_ERROR, this);
        }
```

<u>Listing 3-7</u> shows the `PascalErrorToken` class, which extends class `PascalToken`. It overrides the `extract()` method to do nothing.

<u>Listing 3-7:</u> Class `PascalErrorToken`

```
package wci.frontend.pascal.tokens;

import wci.frontend.*;
import wci.frontend.pascal.*;

import static wci.frontend.pascal.PascalTokenType.*;
```

```
/**
 * <h1>PascalErrorToken</h1>
 *
 * <p>Pascal error token.</p>
 */
public class PascalErrorToken extends PascalToken
{
    /**
     * Constructor.
     * @param source the source from where to fetch subsequent characters.
     * @param errorCode the error code.
     * @param tokenText the text of the erroneous token.
     * @throws Exception if an error occurred.
     */
    public PascalErrorToken(Source source, PascalErrorCode errorCode,
                            String tokenText)
        throws Exception
    {
        super(source);

        this.text = tokenText;
        this.type = ERROR;
        this.value = errorCode;
    }

    /**
     * Do nothing.  Do not consume any source characters.
     * @throws Exception if an error occurred.
     */
    protected void extract()
        throws Exception
    {
    }
}
```

[Listing 3-4](#) showed some error handling:

```
035 {Bad tokens}
036 123e99  123456789012345  1234.56E.  3.  5..  .14  314.e-2
        ^
*** Real literal out of range [at "123e99"]
             ^
*** Integer literal out of range [at "123456789012345"]
                            ^
*** Invalid number [at "1234.56E"]
>>> DOT           line=036, pos=33, text="."
                                ^
*** Invalid number [at "3."]
>>> INTEGER       line=036, pos=40, text="5"
>>>               value=5
>>> DOT_DOT       line=036, pos=41, text=".."
>>> DOT           line=036, pos=45, text="."
>>> INTEGER       line=036, pos=46, text="14"
>>>               value=14
                                      ^
*** Invalid number [at "314."]
>>> IDENTIFIER    line=036, pos=54, text="e"
>>> MINUS         line=036, pos=55, text="-"
>>> INTEGER       line=036, pos=56, text="2"
>>>               value=2
037 What?
>>> IDENTIFIER    line=037, pos= 0, text="What"
          ^
*** Invalid character [at "?"]
038 &apos;String not closed
        ^
*** Unexpected end of file [at "&apos;String not closed "]
```

Whenever the `messageReceived()` method of the inner class `ParserMessageListener` of the `Pascal` main class (see [Listing 3-2](#)) receives a SYNTAX_ERROR message, it displays a ^ character that points to the first character of the erroneous token and the error message text.

How to Scan for Tokens

Suppose a Pascal program contains the source line

```
    IF (index >= 10) THEN
```

How would a scanner go about scanning for and extracting the tokens contained in the line? Assume that the line is indented by four leading blanks and the current position is 0, as indicated below by the bold box around the first character:

The scanner reads and skips over the leading blanks up to the first nonblank character. The current character is now the letter I:

Since it is a letter, the I tells the scanner that the next token must be a word. It extracts a word token by reading and copying characters up to but not including the first character that is not valid for a word, which in this case is a blank. The scanner determines that the word is the reserved word IF. After the scanner has constructed the word token, the current character is the blank after the F:
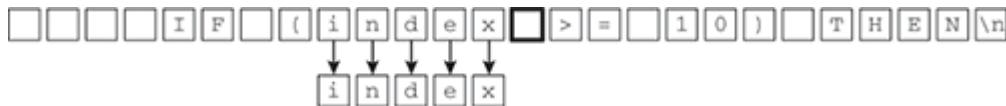
The scanner reads and skips over any blanks between tokens:

The (tells the scanner that it must next extract a special symbol token. Afterwards, the current character is the i:

`IF (index >= 10) THEN \n` — current character **i**, extracts `(`

The i tells the scanner that it must next extract a word token. As before, it reads and copies characters up to but not including the first invalid character for a word. The scanner determines that `index` is an identifier. The current character is now the blank after the x:

`IF (index >= 10) THEN \n` — reads `i n d e x`

The scanner reads and skips the blank:

`IF (index >= 10) THEN \n` — current character **>**

The scanner sees the >, and extracts the >= special symbol token:

`IF (index >= 10) THEN \n` — extracts `> =`

After reading and skipping the blank:

`IF (index >= 10) THEN \n` — current character **1**

The scanner sees the 1 which tells it that it must next extract a number token by reading and copying characters up to but not including the first character that is not valid for a number, which in this case is a). It determines that the number is an integer and calculates the value of 10:

`IF (index >= 10) THEN \n` — reads `1 0`

The) tells the scanner to extract another special symbol token:

`IF (index >= 10) THEN \n` — extracts `)`

The scanner reads and skips the blank:

`IF (index >= 10) THEN \n` — current character **T**

The T tells the scanner to extract another word token:

`IF (index >= 10) THEN \n` — reads `T H E N`

The scanner determines that `THEN` is a reserved word.

After it is done scanning the line, the scanner has extracted the following tokens:

Type	Text string
Word (reserved word)	IF
Special symbol	(
Word (identifier)	index
Special symbol	>=
Number (integer)	10
Special symbol	)
Word (reserved word)	THEN

Key points from this short example include:

- The scanner reads and skips whitespace characters (such as blanks) between tokens. When it's done, the current character is nonblank.
- This nonblank character determines the type of token the scanner will extract next, and the character becomes the first character of that token.
- The scanner extracts a token by reading and copying successive source characters up to but not including the first character that cannot be part of the token.
- Extracting a token consumes all the source characters that constitute the token. Therefore, after extracting a token, the current character is the first character after the last token character.

A Pascal Scanner

The previous chapter had the beginnings of the `PascalScanner` class in the `frontend.pascal` package. Now expand method `extractToken()` and add a new method `skipWhiteSpace()`. See .

Listing 3-8: Methods **`extractToken()`** and **`skipWhiteSpace()`** of class **`PascalScanner`**

```
/**
 * Extract and return the next Pascal token from the source.
 * @return the next token.
 * @throws Exception if an error occurred.
 */
protected Token extractToken()
    throws Exception
{
    skipWhiteSpace();

    Token token;
    char currentChar = currentChar();

    // Construct the next token.  The current character determines the
    // token type.
    if (currentChar == EOF) {
        token = new EofToken(source, END_OF_FILE);
    }
    else if (Character.isLetter(currentChar)) {
        token = new PascalWordToken(source);
    }
    else if (Character.isDigit(currentChar)) {
        token = new PascalNumberToken(source);
    }
    else if (currentChar == '\'') {
        token = new PascalStringToken(source);
    }
    else if (PascalTokenType.SPECIAL_SYMBOLS
                .containsKey(Character.toString(currentChar))) {
        token = new PascalSpecialSymbolToken(source);
    }
    else {
        token = new PascalErrorToken(source, INVALID_CHARACTER,
                                     Character.toString(currentChar));
        nextChar();  // consume character
    }

    return token;
}

/**
 * Skip whitespace characters by consuming them.  A comment is whitespace.
 * @throws Exception if an error occurred.
 */
private void skipWhiteSpace()
    throws Exception
{
    char currentChar = currentChar();

    while (Character.isWhitespace(currentChar) || (currentChar == '{')) {

        // Start of a comment?
        if (currentChar == '{') {
            do {
                currentChar = nextChar();  // consume comment characters
            } while ((currentChar != '}') && (currentChar != EOF));

            // Found closing '}'?
            if (currentChar == '}') {
                currentChar = nextChar();  // consume the '}'
            }
        }

        // Not a comment.
        else {
            currentChar = nextChar();  // consume whitespace character
        }
    }
}
```

The `extractToken()` method works exactly as described in the example. First, it reads and skips over any whitespace characters. Then the current (non-whitespace) character is the first character of the next token, whose type is determined by that character. The method creates and returns a `PascalWordToken`, a `PascalNumberToken`, a `PascalStringToken`, or a `PascalSpecialSymbolToken` object. Therefore, after `extractToken()` has read the initial character of each token, the new token object reads and copies the remaining characters of the token.

The `extractToken()` method can also create and return an `EofToken` at the end of the source, or a `PascalErrorToken` for an invalid character that cannot be the start of any Pascal token.

Method `skipWhiteSpace()` skips all whitespace characters between tokens, as determined by Java's `Character.isWhitespace()` method. It also skips over entire Pascal comments, since each comment is equivalent to a blank.

Pascal Tokens

In the previous chapter, you defined the language-independent `TokenType` marker interface. Now define an enumerated type `PascalTokenType` whose enumerated values represent all the Pascal tokens. shows that it does indeed implement `TokenType`.

Listing 3-9: The **`PascalTokenType`** enumerated type

```
package wci.frontend.pascal;

import java.util.Hashtable;
import java.util.HashSet;
```

```java
import wci.frontend.TokenType;

/**
 * <h1>PascalTokenType</h1>
 *
 * <p>Pascal token types.</p>
 */
public enum PascalTokenType implements TokenType
{
    // Reserved words.
    AND, ARRAY, BEGIN, CASE, CONST, DIV, DO, DOWNTO, ELSE, END,
    FILE, FOR, FUNCTION, GOTO, IF, IN, LABEL, MOD, NIL, NOT,
    OF, OR, PACKED, PROCEDURE, PROGRAM, RECORD, REPEAT, SET,
    THEN, TO, TYPE, UNTIL, VAR, WHILE, WITH,

    // Special symbols.
    PLUS("+"), MINUS("-"), STAR("*"), SLASH("/"), COLON_EQUALS(":="),
    DOT("."), COMMA(","), SEMICOLON(";"), COLON(":"), QUOTE("&apos;"),
    EQUALS("="), NOT_EQUALS("<>"), LESS_THAN("<"), LESS_EQUALS("<="),
    GREATER_EQUALS(">="), GREATER_THAN(">"), LEFT_PAREN("("), RIGHT_PAREN(")"),
    LEFT_BRACKET("["), RIGHT_BRACKET("]"), LEFT_BRACE("{"), RIGHT_BRACE("}"),
    UP_ARROW("^"), DOT_DOT(".."),

    IDENTIFIER, INTEGER, REAL, STRING,
    ERROR, END_OF_FILE;

    private static final int FIRST_RESERVED_INDEX = AND.ordinal();
    private static final int LAST_RESERVED_INDEX  = WITH.ordinal();

    private static final int FIRST_SPECIAL_INDEX = PLUS.ordinal();
    private static final int LAST_SPECIAL_INDEX  = DOT_DOT.ordinal();

    private String text;  // token text

    /**
     * Constructor.
     */
    PascalTokenType()
    {
        this.text = this.toString().toLowerCase();
    }

    /**
     * Constructor.
     * @param text the token text.
     */
    PascalTokenType(String text)
    {
        this.text = text;
    }

    /**
     * Getter.
     * @return the token text.
     */
    public String getText()
    {
        return text;
    }

    // Set of lower-cased Pascal reserved word text strings.
    public static HashSet<String> RESERVED_WORDS = new HashSet<String>();
    static {
        PascalTokenType values[] = PascalTokenType.values();
        for (int i = FIRST_RESERVED_INDEX; i <= LAST_RESERVED_INDEX; ++i) {
            RESERVED_WORDS.add(values[i].getText().toLowerCase());
        }
    }

    // Hash table of Pascal special symbols.  Each special symbol&apos;s text
    // is the key to its Pascal token type.
    public static Hashtable<String, PascalTokenType> SPECIAL_SYMBOLS =
        new Hashtable<String, PascalTokenType>();
    static {
        PascalTokenType values[] = PascalTokenType.values();
        for (int i = FIRST_SPECIAL_INDEX; i <= LAST_SPECIAL_INDEX; ++i) {
            SPECIAL_SYMBOLS.put(values[i].getText(), values[i]);
        }
    }
}
```

In <u>Listing 3-1</u>, class `PascalParserTD` uses the enumerated value `ERROR` in the `parse()` method. In <u>Listing 3-2</u>, inner class `ParserMessageListener` of the main `Pascal` class uses the enumerated value `STRING` while processing `TOKEN` messages.

The static set `RESERVED_WORDS` contains the text strings of all the Pascal reserved words. Use this set later in class `PascalWordToken` to determine whether a word is a reserved word or an identifier. Note that because the enumerated value for each reserved word is named after the reserved word, the value's text is the reserved word's text string. For example, the value of the enumerated value `BEGIN` is the string "`BEGIN`". Also, since Pascal words are case insensitive, "normalize" the words in this set by setting them all to lowercase.

The static hash table `SPECIAL_SYMBOLS` contains one entry for each Pascal special symbol. The entry's key is the special symbol's text string, as set by the constructor for the enumerated value, and the entry's value is the enumerated value itself. For example, the entry whose key is "`:=`" has the value `COLON_EQUALS`. Class `PascalScanner` (see <u>Listing 3-8</u>) uses this hash table to determine whether or not to create and return a `PascalSpecialSymbolToken`. Class `PascalSpecialSymbolToken` will also use this hash table.

<u>Listing 3-10</u> shows the token subclass `PascalToken`, which in turn is the base class of all the Pascal token subclasses. It extends the language-independent framework class `Token`. Although it currently does not add any fields or methods, it provides the flexibility to do so in the future.

<u>Listing 3-10:</u> Class `PascalToken`

```java
package wci.frontend.pascal;

import wci.frontend.*;

/**
 * <h1>PascalToken</h1>
 *
 * <p>Base class for Pascal token classes.</p>
 */
public class PascalToken extends Token
{
    /**
```

```
     * Constructor.
     * @param source the source from where to fetch the token&apos;s characters.
     * @throws Exception if an error occurred.
     */
    protected PascalToken(Source source)
        throws Exception
    {
        super(source);
    }
}
```
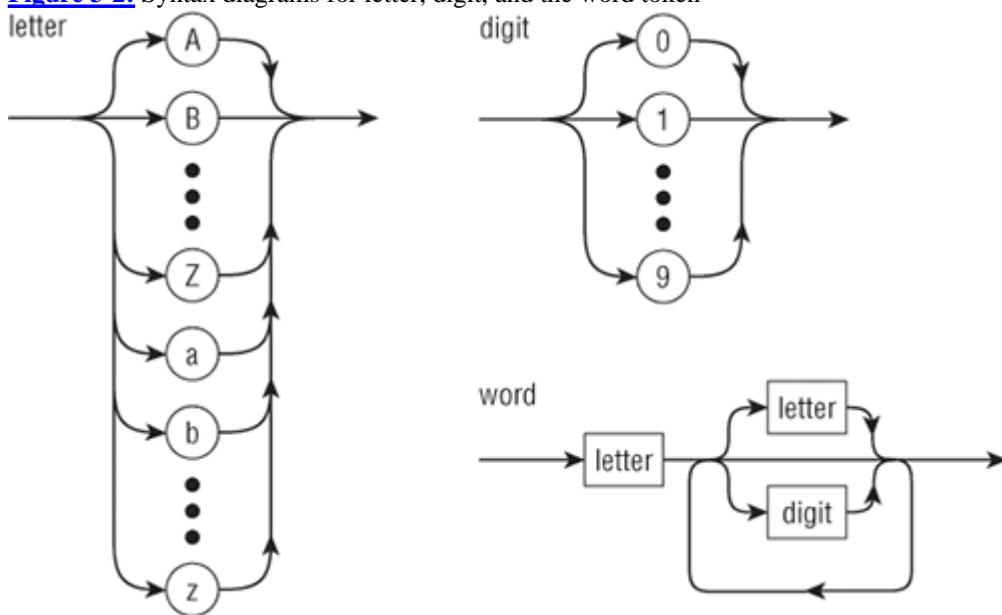
Syntax Diagrams

Develop the remaining subclasses of `PascalToken` in package `frontend.pascal.tokens` to extract the various types of Pascal tokens. But first, you need a good specification of the syntax of these language elements. There are several common ways to do this, but Pascal's relatively simple syntax lends itself very well to using *syntax diagrams*, which are graphical representations of a language's syntax rules.[2]

[2] The last chapter in this book presents a textual way of representing a language's syntax.

[Figure 3-2](#) shows three diagrams. The first one specifies that a letter can be any character from uppercase `A` through uppercase `Z` or lowercase `a` through lowercase `z`. The second diagram specifies that a digit can be any character from `0` through `9`. The third diagram specifies that a word token is a single letter followed by a sequence of zero or more letters and digits.

[Figure 3-2:](#) Syntax diagrams for letter, digit, and the word token

Design Note

Syntax diagrams are easy to read: Just follow the paths in the directions indicated by the arrows. Some paths fork to indicate choices: A letter can be an `A` or a `B` or a `c`, etc. Other paths loop back to indicate a sequence: After the initial letter in a word token, there is a sequence of zero or more letters and digits.

Rounded boxes represent literal text, such as the letter `A` or the digit `0`. (Starting in Chapter 5, you'll also use rounded boxes to represent the literal text of reserved words such as `AND`, `OR`, and `MOD`.) A rectangular box is a reference to another diagram. For example, the diagram for the word token refers to the diagrams for letter and digit. In more formal terms, a rounded box represents a terminal symbol, and a rectangular box represents a nonterminal symbol.

Word Tokens

Method `extractToken()` of class `PascalScanner` (see [Listing 3-8](#)) creates a new Pascal word token only when the current character is a letter:

```
else if (Character.isLetter(currentChar)) {
    token = new PascalWordToken(source);
}
```

Class `PascalWordToken` in package `frontend.pascal.tokens` is a subclass of `PascalToken`. [Listing 3-11](#) shows its `extract()` method.

[Listing 3-11:](#) Method `extract()` of class `PascalWordToken`

```
/**
 * Extract a Pascal word token from the source.
 * @throws Exception if an error occurred.
 */
protected void extract()
    throws Exception
{
    StringBuilder textBuffer = new StringBuilder();
    char currentChar = currentChar();

    // Get the word characters (letter or digit).  The scanner has
    // already determined that the first character is a letter.
    while (Character.isLetterOrDigit(currentChar)) {
        textBuffer.append(currentChar);
        currentChar = nextChar();  // consume character
    }
```

```
        text = textBuffer.toString();

        // Is it a reserved word or an identifier?
        type = (RESERVED_WORDS.contains(text.toLowerCase()))
                ? PascalTokenType.valueOf(text.toUpperCase())  // reserved word
                : IDENTIFIER;                                  // identifier
    }
```

Method `extract()` implements the syntax rules for a Pascal word token as shown in the syntax diagram in Figure 3-2. After taking the initial letter, it consumes any subsequent letters and digits to build the word token's text string. When it's done, the value of `currentChar` is the first character that is *not* a letter or digit and therefore that character is the first one after the token.

Afterwards, `extract()` determines whether the word is a reserved word. If the word's text string is in the `TokenTypes.RESERVED_WORDS` set (see Listing 3-9), it must be a reserved word. Since Pascal reserved words are case insensitive, the test for set membership is done in lowercase. If the word is indeed a reserved word, the token's type is the corresponding `PascalTokenType` enumerated value. Otherwise, the token type is `IDENTIFIER`. For now, leave the value of a word token to be null.

Listing 3-4 showed some of the results of class `PascalWordToken`:

```
009 {Word tokens}
010 Hello world
>>> IDENTIFIER       line=010, pos= 0, text="Hello"
>>> IDENTIFIER       line=010, pos= 6, text="world"
011 begin BEGIN Begin BeGiN begins
>>> BEGIN            line=011, pos= 0, text="begin"
>>> BEGIN            line=011, pos= 6, text="BEGIN"
>>> BEGIN            line=011, pos=12, text="Begin"
>>> BEGIN            line=011, pos=18, text="BeGiN"
>>> IDENTIFIER       line=011, pos=24, text="begins"
```
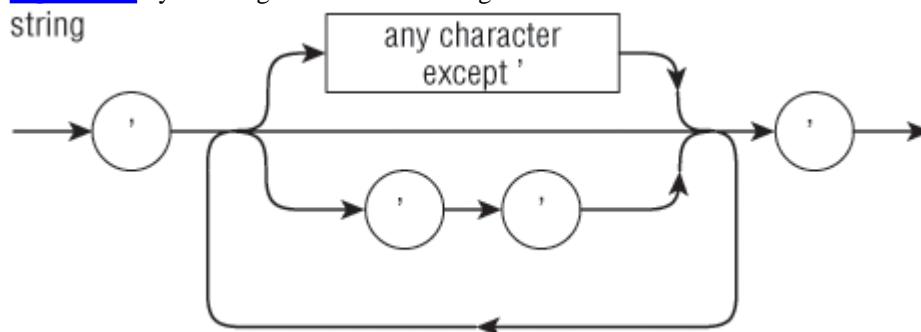
Source line 011 contains the reserved word `BEGIN` four times to verify that `PascalWordToken` is case insensitive. Of course, the token type of `begins` is `IDENTIFIER`.

String Tokens

Figure 3-3 shows the syntax diagram for Pascal string tokens.

Figure 3-3: Syntax diagram for Pascal string tokens

A Pascal string begins and ends with a single-quote character. Between the single-quotes is a sequence of zero or more characters that comprise the string token's value. Within the character sequence, two immediately adjacent single-quotes represent one single-quote character that is part of the string value.

Class `PascalScanner` (see Listing 3-8) constructs a `PascalStringToken` when the current character is a single-quote:

```
        else if (currentChar == '\'') {
            token = new PascalStringToken(source);
        }
```

Listing 3-12 shows the `extract()` method of the Pascal token subclass `PascalStringToken`.

Listing 3-12: Method `extract()` of class `PascalStringToken`

```
/**
 * Extract a Pascal string token from the source.
 * @throws Exception if an error occurred.
 */
protected void extract()
    throws Exception
{
    StringBuilder textBuffer = new StringBuilder();
    StringBuilder valueBuffer = new StringBuilder();

    char currentChar = nextChar();  // consume initial quote
    textBuffer.append('\'');

    // Get string characters.
    do {
        // Replace any whitespace character with a blank.
        if (Character.isWhitespace(currentChar)) {
            currentChar = ' ';
        }

        if ((currentChar != '\'') && (currentChar != EOF)) {
            textBuffer.append(currentChar);
            valueBuffer.append(currentChar);
            currentChar = nextChar();  // consume character
        }

        // Quote?  Each pair of adjacent quotes represents a single-quote.
        if (currentChar == '\'') {
            while ((currentChar == '\'') && (peekChar() == '\'')) {
                textBuffer.append("''");
                valueBuffer.append(currentChar); // append single-quote
                currentChar = nextChar();        // consume pair of quotes
                currentChar = nextChar();
            }
```

```
        }
    } while ((currentChar != '\'') && (currentChar != EOF));

    if (currentChar == '\'') {
        nextChar();  // consume final quote
        textBuffer.append('\'');

        type = STRING;
        value = valueBuffer.toString();
    }
    else {
        type = ERROR;
        value = UNEXPECTED_EOF;
    }

    text = textBuffer.toString();
}
```

Method `extract()` consumes the string's characters. It replaces any whitespace character (such as an end-of-line character) with a single blank, and it must check for an unexpected end of the source file. Whenever the method encounters a single-quote character, it calls `peekChar()` to determine whether the character is the ending single-quote of the string or the first of a pair of adjacent single-quote characters. In the latter case, the method's inner loop consumes pairs of single-quote characters and appends one single-quote character to the string value for each pair. If the source file unexpectedly ends, the method sets the token's type to ERROR and the value to the `PascalErrorCode` enumerated value `UNEXPECTED_EOF`.

[Listing 3-4](#) showed some of the results of class `PascalStringToken`:

```
013 {String tokens}
014 'Hello, world.'
>>> STRING           line=014, pos= 0, text="'Hello, world.'"
>>>                  value="Hello, world."
015 'It''s Friday!'
>>> STRING           line=015, pos= 0, text="'It''s Friday!'"
>>>                  value="It's Friday!"
016 ''
>>> STRING           line=016, pos= 0, text="''"
>>>                  value=""
017 ' '' ''' ''   ''''''''''
>>> STRING           line=017, pos= 0, text="' '' ''' ''   '"
>>>                  value=" ' ' "
>>> STRING           line=017, pos=12, text="'''''''''"
>>>                  value="''''"
018 'This string
019 spans
020 source lines.'
>>> STRING           line=018, pos= 0, text="'This string spans source lines.'"
>>>                  value="This string spans source lines."
```

`PascalStringToken` properly handles the empty string (source line 016) and pairs of adjacent single-quote characters properly (especially source line 017). It replaces each end-of-line character within the string with a blank (source lines 018, 019, and 020).

Special Symbol Tokens

Class `PascalScanner` (see [Listing 3-8](#)) constructs a `PascalSpecialSymbolToken` when the current character is one of the values contained in the `PascalTokenType` .SPECIAL_SYMBOLS hash table (see [Listing 3-9](#)):

```
else if (PascalTokenType.SPECIAL_SYMBOLS
        .containsKey(Character.toString(currentChar))) {
    token = new PascalSpecialSymbolToken(source);
}
```

[Listing 3-13](#) shows method `extract()` of the Pascal token subclass `PascalSpecialSymbolToken`.

[Listing 3-13:](#) Method `extract()` of class `PascalSpecialSymbolToken`

```
/**
 * Extract a Pascal special symbol token from the source.
 * @throws Exception if an error occurred.
 */
protected void extract()
    throws Exception
{
    char currentChar = currentChar();

    text = Character.toString(currentChar);
    type = null;

    switch (currentChar) {

        // Single-character special symbols.
        case '+':  case '-':  case '*':  case '/':  case ',':
        case ';':  case '\'': case '=':  case '(':  case ')':
        case '[':  case ']':  case '{':  case '}':  case '^': {
            nextChar();  // consume character
            break;
        }

        // : or :=
        case ':': {
            currentChar = nextChar();  // consume ':';

            if (currentChar == '=') {
                text += currentChar;
                nextChar();  // consume '='
            }

            break;
        }

        // < or <= or <>
        case '<': {
            currentChar = nextChar();  // consume '<';

            if (currentChar == '=') {
                text += currentChar;
                nextChar();  // consume '='
            }
```

```
            else if (currentChar == '>') {
                text += currentChar;
                nextChar();  // consume '>'
            }

            break;
        }

        // > or >=
        case '>': {
            currentChar = nextChar();  // consume '>';

            if (currentChar == '=') {
                text += currentChar;
                nextChar();  // consume '='
            }

            break;
        }

        // . or ..
        case '.': {
            currentChar = nextChar();  // consume '.';

            if (currentChar == '.') {
                text += currentChar;
                nextChar();  // consume '.'
            }

            break;
        }

        default: {
            nextChar();  // consume bad character
            type = ERROR;
            value = INVALID_CHARACTER;
        }
    }

    // Set the type if it wasn't an error.
    if (type == null) {
        type = SPECIAL_SYMBOLS.get(text);
    }
}
```

The `extract()` method attempts to extract a special symbol token and consume the token characters. A Pascal special symbol token consists of either one or two characters. If successful, the method uses the `SPECIAL_SYMBOLS` hash table to set the token's type to the correct enumerated value. Otherwise, if there was an error, the method sets the token's type to `ERROR` and its value to the `PascalErrorCode` enumerated value `INVALID_CHARACTER`.

showed some of the results of class `PascalSpecialSymbolToken`:

```
022 {Special symbol tokens}
023 + - * / := . , ; : = <> < <= >= > ( ) [ ] { } ^ ..
>>> PLUS            line=023, pos= 0, text="+"
>>> MINUS           line=023, pos= 2, text="-"
>>> STAR            line=023, pos= 4, text="*"
>>> SLASH           line=023, pos= 6, text="/"
>>> COLON_EQUALS    line=023, pos= 8, text=":="
>>> DOT             line=023, pos=11, text="."
>>> COMMA           line=023, pos=13, text=","
>>> SEMICOLON       line=023, pos=15, text=";"
>>> COLON           line=023, pos=17, text=":"
>>> EQUALS          line=023, pos=19, text="="
>>> NOT_EQUALS      line=023, pos=21, text="<>"
>>> LESS_THAN       line=023, pos=24, text="<"
>>> LESS_EQUALS     line=023, pos=26, text="<="
>>> GREATER_EQUALS  line=023, pos=29, text=">="
>>> GREATER_THAN    line=023, pos=32, text=">"
>>> LEFT_PAREN      line=023, pos=34, text="("
>>> RIGHT_PAREN     line=023, pos=36, text=")"
>>> LEFT_BRACKET    line=023, pos=38, text="["
>>> RIGHT_BRACKET   line=023, pos=40, text="]"
>>> RIGHT_BRACE     line=023, pos=46, text="}"
>>> UP_ARROW        line=023, pos=48, text="^"
>>> DOT_DOT         line=023, pos=50, text=".."
024 +-:=<>=<==.....
>>> PLUS            line=024, pos= 0, text="+"
>>> MINUS           line=024, pos= 1, text="-"
>>> COLON_EQUALS    line=024, pos= 2, text=":="
>>> NOT_EQUALS      line=024, pos= 4, text="<>"
>>> EQUALS          line=024, pos= 6, text="="
>>> LESS_EQUALS     line=024, pos= 7, text="<="
>>> EQUALS          line=024, pos= 9, text="="
>>> DOT_DOT         line=024, pos=10, text=".."
>>> DOT_DOT         line=024, pos=12, text=".."
>>> DOT             line=024, pos=14, text="."
```

Number Tokens

Figure 3-4 shows the syntax diagram for Pascal number tokens.

An unsigned integer is a sequence of digits. A Pascal integer number token is an unsigned integer. A Pascal real number token begins with an unsigned integer (the whole part), which is followed by either:
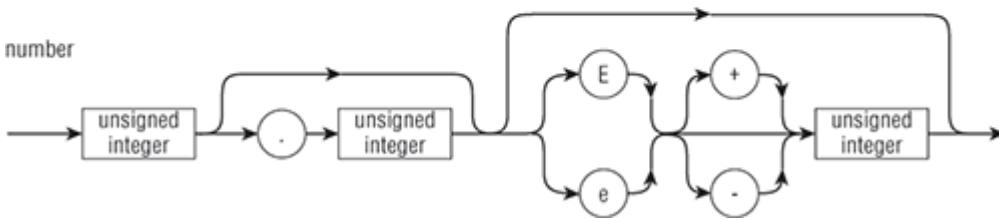
- A decimal point followed by an unsigned integer (the fraction part), or
- An `E` or `e`, optionally followed by `+` or `-`, followed by an unsigned integer (the exponent part), or
- A fraction part followed by an exponent part.

Figure 3-4: Syntax diagram for Pascal number tokens

Class `PascalScanner` (see Listing 3-8) constructs a `PascalNumberToken` when the current character is a digit:

```
else if (Character.isDigit(currentChar)) {
    token = new PascalNumberToken(source);
}
```

Listing 3-14 shows method `extract()` of class `PascalNumberToken`, which is the most complicated of all the `PascalToken` subclasses.

Listing 3-14: Method `extract()` of class `PascalNumberToken`

```
/**
 * Extract a Pascal number token from the source.
 * @throws Exception if an error occurred.
 */
protected void extract()
    throws Exception
{
    StringBuilder textBuffer = new StringBuilder();  // token's characters
    extractNumber(textBuffer);
    text = textBuffer.toString();
}
```

Method `extract()` calls `extractNumber()`, which does most of the work. See Listing 3-15.

Listing 3-15: Method `extractNumber()` of class `PascalNumberToken`

```
/**
 * Extract a Pascal number token from the source.
 * @param textBuffer the buffer to append the token's characters.
 * @throws Exception if an error occurred.
 */
protected void extractNumber(StringBuilder textBuffer)
    throws Exception
{
    String wholeDigits = null;      // digits before the decimal point
    String fractionDigits = null;   // digits after the decimal point
    String exponentDigits = null;   // exponent digits
    char exponentSign = '+';        // exponent sign '+' or '-'
    boolean sawDotDot = false;      // true if saw .. token
    char currentChar;               // current character

    type = INTEGER;  // assume INTEGER token type for now

    // Extract the digits of the whole part of the number.
    wholeDigits = unsignedIntegerDigits(textBuffer);
    if (type == ERROR) {
        return;
    }

    // Is there a . ?
    // It could be a decimal point or the start of a .. token.
    currentChar = currentChar();
    if (currentChar == '.') {
        if (peekChar() == '.') {
            sawDotDot = true;  // it's a ".." token, so don't consume it
        }
        else {
            type = REAL;  // decimal point, so token type is REAL
            textBuffer.append(currentChar);
            currentChar = nextChar();  // consume decimal point

            // Collect the digits of the fraction part of the number.
            fractionDigits = unsignedIntegerDigits(textBuffer);
            if (type == ERROR) {
                return;
            }
        }
    }

    // Is there an exponent part?
    // There cannot be an exponent if we already saw a ".." token.
    currentChar = currentChar();
    if (!sawDotDot && ((currentChar == 'E') || (currentChar == 'e'))) {
        type = REAL;  // exponent, so token type is REAL
        textBuffer.append(currentChar);
        currentChar = nextChar();  // consume 'E' or 'e'

        // Exponent sign?
        if ((currentChar == '+') || (currentChar == '-')) {
            textBuffer.append(currentChar);
            exponentSign = currentChar;
            currentChar = nextChar();  // consume '+' or '-'
        }

        // Extract the digits of the exponent.
        exponentDigits = unsignedIntegerDigits(textBuffer);
    }

    // Compute the value of an integer number token.
    if (type == INTEGER) {
```

```
            int integerValue = computeIntegerValue(wholeDigits);

            if (type != ERROR) {
                value = new Integer(integerValue);
            }
        }

        // Compute the value of a real number token.
        else if (type == REAL) {
            float floatValue = computeFloatValue(wholeDigits, fractionDigits,
                                      exponentDigits, exponentSign);

          if (type != ERROR) {
              value = new Float(floatValue);
          }
        }
    }
```

String fields `wholeDigits`, `fractionDigits`, and `exponentDigits` will contain the sequences of digits before the decimal point (the whole part of the number), after the decimal point (the fraction part), and after the 'E' or 'e' (the exponent part), respectively. As shown by the syntax diagram in Figure 3-4, fields `fractionDigits` and `exponentDigits` could remain null. The method initially assumes that the token type is INTEGER and changes it to REAL only if there is a fraction part or an exponent part. It extracts the digits in the different parts of the number by calling method `unsignedIntegerDigits()`, which makes sure that there is at least one digit. See Listing 3-16.

Listing 3-16: Method `unsignedIntegerDigits()` of class `PascalNumberToken`

```
/**
 * Extract and return the digits of an unsigned integer.
 * @param textBuffer the buffer to append the token's characters.
 * @return the string of digits.
 * @throws Exception if an error occurred.
 */
private String unsignedIntegerDigits(StringBuilder textBuffer)
    throws Exception
{
    char currentChar = currentChar();

    // Must have at least one digit.
    if (!Character.isDigit(currentChar)) {
        type = ERROR;
        value = INVALID_NUMBER;
        return null;
    }

    // Extract the digits.
    StringBuilder digits = new StringBuilder();
    while (Character.isDigit(currentChar)) {
        textBuffer.append(currentChar);
        digits.append(currentChar);
        currentChar = nextChar();  // consume digit
    }

    return digits.toString();
}
```

If method `extractNumber()` encounters the . character after the whole part, it cannot immediately assume that it is a decimal point, because it might be the start of the .. token. A call to `peekChar()` to look ahead one character determines which is the case. After the method has extracted all the parts of a number, it calls either `computeIntegerValue()` or `computeFloatValue()` to compute the number's value depending on whether the type is INTEGER or REAL. See Listings 3-17 and 3-18.

Listing 3-17: Method `computeIntegerValue()` of class `PascalNumberToken`

```
/**
 * Compute and return the integer value of a string of digits.
 * Check for overflow.
 * @param digits the string of digits.
 * @return the integer value.
 */
private int computeIntegerValue(String digits)
{
    // Return 0 if no digits.
    if (digits == null) {
        return 0;
    }

    int integerValue = 0;
    int prevValue = -1;    // overflow occurred if prevValue > integerValue
    int index = 0;

    // Loop over the digits to compute the integer value
    // as long as there is no overflow.
    while ((index < digits.length()) && (integerValue >= prevValue)) {
        prevValue = integerValue;
        integerValue = 10*integerValue +
                    Character.getNumericValue(digits.charAt(index++));
    }

    // No overflow:  Return the integer value.
    if (integerValue >= prevValue) {
        return integerValue;
    }

    // Overflow:  Set the integer out of range error.
    else {
        type = ERROR;
        value = RANGE_INTEGER;
        return 0;
    }
}
```

Method `computeIntegerValue()` computes the integer value of a string of digits. It checks for an overflow by making sure the value doesn't "wrap around" and become less than the previous value. If there was an overflow, the method sets the token's type to ERROR and its value to the `PascalErrorCode` enumerated value `RANGE_INTEGER`.

Listing 3-18: Method `computeFloatValue()` of class `PascalNumberToken`

```
        /**
         * Compute and return the float value of a real number.
         * @param wholeDigits the string of digits before the decimal point.
         * @param fractionDigits the string of digits after the decimal point.
         * @param exponentDigits the string of exponent digits.
         * @param exponentSign the exponent sign.
         * @return the float value.
         */
        private float computeFloatValue(String wholeDigits, String fractionDigits,
                                        String exponentDigits, char exponentSign)
        {
            double floatValue = 0.0;
            int exponentValue = computeIntegerValue(exponentDigits);
            String digits = wholeDigits;  // whole and fraction digits

            // Negate the exponent if the exponent sign is '-'.
            if (exponentSign == '-') {
                exponentValue = -exponentValue;
            }

            // If there are any fraction digits, adjust the exponent value
            // and append the fraction digits.
            if (fractionDigits != null) {
                exponentValue -= fractionDigits.length();
                digits += fractionDigits;
            }

            // Check for a real number out of range error.
            if (Math.abs(exponentValue + wholeDigits.length()) > MAX_EXPONENT) {
                type = ERROR;
                value = RANGE_REAL;
                return 0.0f;
            }

            // Loop over the digits to compute the float value.
            int index = 0;
            while (index < digits.length()) {
                floatValue = 10*floatValue +
                                Character.getNumericValue(digits.charAt(index++));
            }

            // Adjust the float value based on the exponent value.
            if (exponentValue != 0) {
                floatValue *= Math.pow(10, exponentValue);
            }

            return (float) floatValue;
        }
```

Method `computeFloatValue()` computes a `float` value from the digit strings representing the whole, fraction, and exponent parts, along with the exponent sign. It calls `computeIntegerValue()` to compute the integer value of the exponent, which it negates if the exponent sign is `-`. If there were any fraction digits, the method further adjusts the exponent value by subtracting the length of the string of fraction digits, if any. Finally, if the sum of the adjusted exponent value and the number of whole digits exceeds MAX_EXPONENT, the number is out of range, and the method sets the token's type to ERROR and its value to the `PascalErrorCode` enumerated value RANGE_REAL.[3]

[3] The value of MAX_EXPONENT depends on the underlying machine architecture and the implemented floating-point standard. The minimum exponent value is not necessarily the negative of the maximum exponent value.

[4] Our algorithm for computing float values generates a small roundoff error.

The method computes the value of the concatenation of the whole and any fraction digits. Multiplying this value by a call to Java's `Math.pow()` with the adjusted exponent value computes the final value of the number token.

Here is an example of how method `extractNumber()` computes the value of a number token. For the token string is `"31415.926e-4,"` method `extractNumber()` passes the following parameter values to method `computeFloatValue()`:

```
wholeDigits:    "31415"
fractionDigits: "926"
exponentDigits: "4"
exponentSign:   '-'
```

Method `computeFloatValue()` calls `computeIntegerValue()` to compute the value of `exponentValue`, which it then adjusts twice, first to negate it since `exponentSign` is `-` and second to subtract the length 3 of the `fractionDigits` string:

```
exponentValue:  4 as computed by computeIntegerValue()
exponentValue: -4 after negation since exponentSign is '-'
exponentValue: -7 after subtracting fractionDigits.length()
```

Finally, `computeFloatValue()` computes the value of the concatenated digit string `wholeDigits + fractionDigits`, giving 31415926.0, which it adjusts by multiplying by `Math.pow(10, exponentValue)`. This results in 3.1415926 as the final float value of the number token.[4]

What about a leading + or - sign, as in `-54` or `+3.1415926`? You'll see in Chapter 5 that the parser considers such a leading sign to be a separate token.

showed some of the results of class `PascalNumberToken`:

```
026 {Number tokens}
027 0 1 20 00000000000000000032  31415926
>>> INTEGER        line=027, pos= 0, text="0"
>>>                value=0
>>> INTEGER        line=027, pos= 2, text="1"
>>>                value=1
>>> INTEGER        line=027, pos= 4, text="20"
>>>                value=20
>>> INTEGER        line=027, pos= 7, text="00000000000000000032"
>>>                value=32
>>> INTEGER        line=027, pos=29, text="31415926"
>>>                value=31415926
028 3.1415926  3.14159265358979323384626433
>>> REAL           line=028, pos= 0, text="3.1415926"
>>>                value=3.1415925
>>> REAL           line=028, pos=11, text="3.14159265358979323384626433"
>>>                value=3.1415927
029 0.00031415926E4  0.00031415926e+00004  31415.926e-4
>>> REAL           line=029, pos= 0, text="0.00031415926E4"
>>>                value=3.1415925
>>> REAL           line=029, pos=17, text="0.00031415926e+00004"
```

```
>>>              value=3.1415925
>>> REAL         line=029, pos=39, text="31415.926e-4"
>>>              value=3.1415925
030 3141592600000000000000000000000e-30
>>> REAL         line=030, pos= 0, text="3141592600000000000000000000000e-30"
>>>              value=3.1415925
```

Design Note

The scanner is a major component of the front end of a compiler or an interpreter, and there was a significant amount of code in this chapter. But you followed the Strategy Design Pattern and implemented subclasses of `PascalToken` so that each one knows how to extract one type of Pascal token from the source program.

Because each of these subclasses has methods that together have only one responsibility, each subclass has high cohesion. For example, the methods of `PascalNumberToken` are responsible only for extracting number tokens. Highly cohesive classes are loosely coupled and easier to maintain. If you decide, say, to improve the way the scanner computes the value of a real number in order to reduce roundoff errors, you would only need to make changes to `PascalNumberToken`.

Had you made the design decision to make class `PascalToken` extract all the Pascal token types by itself, that class would have very low cohesion. It would then be harder to change how to extract one token type without affecting the other types.

Chapter 4

The Symbol Table

The parser of a compiler or an interpreter builds and maintains a symbol table throughout the translation process as part of semantic analysis. The symbol table stores information about the source program's tokens, mostly the identifiers. As you saw in Figures 1-3 and 2-1, the symbol table is a key intermediate component in the interface between the front and back ends.

Goals and Approach

Maintaining a well-organized symbol table is an important skill for all compiler writers. As a compiler or an interpreter translates a source program, it must be able to enter new information and access and update existing information quickly and efficiently. Otherwise, the translation process slows down or worse, generates incorrect results.

The goals of this chapter are:

- A flexible, language-independent symbol table.
- A simple utility program that parses a Pascal source program and generates a cross-reference listing of its identifiers.

The approach is to create the conceptual design for the symbol table, then develop Java interfaces that represent the design, and finally write Java classes that implement the interfaces. The cross-reference utility program will help verify that your code is correct. It will exercise the symbol table by entering, finding, and updating information.

Symbol Table Conceptual Design

During the translation process, the compiler or interpreter creates and updates entries in the symbol table to contain information about certain tokens in the source program. Each entry has a name, which is the token's text string. For example, an entry for an identifier token uses the identifier's name. The entry also contains information about the identifier. As it translates the source program, the compiler or interpreter looks up and updates this information.

What information should you keep in the symbol table? Any information that is useful! The symbol table entry for an identifier will typically include its type, structure, and how it was defined. One of the goals is to keep the symbol table flexible and not limited to Pascal-specific information. No matter what information the symbol table stores, the basic operations it must support are:

- Enter new information.
- Look up existing information.
- Update existing information.

The Symbol Table Stack
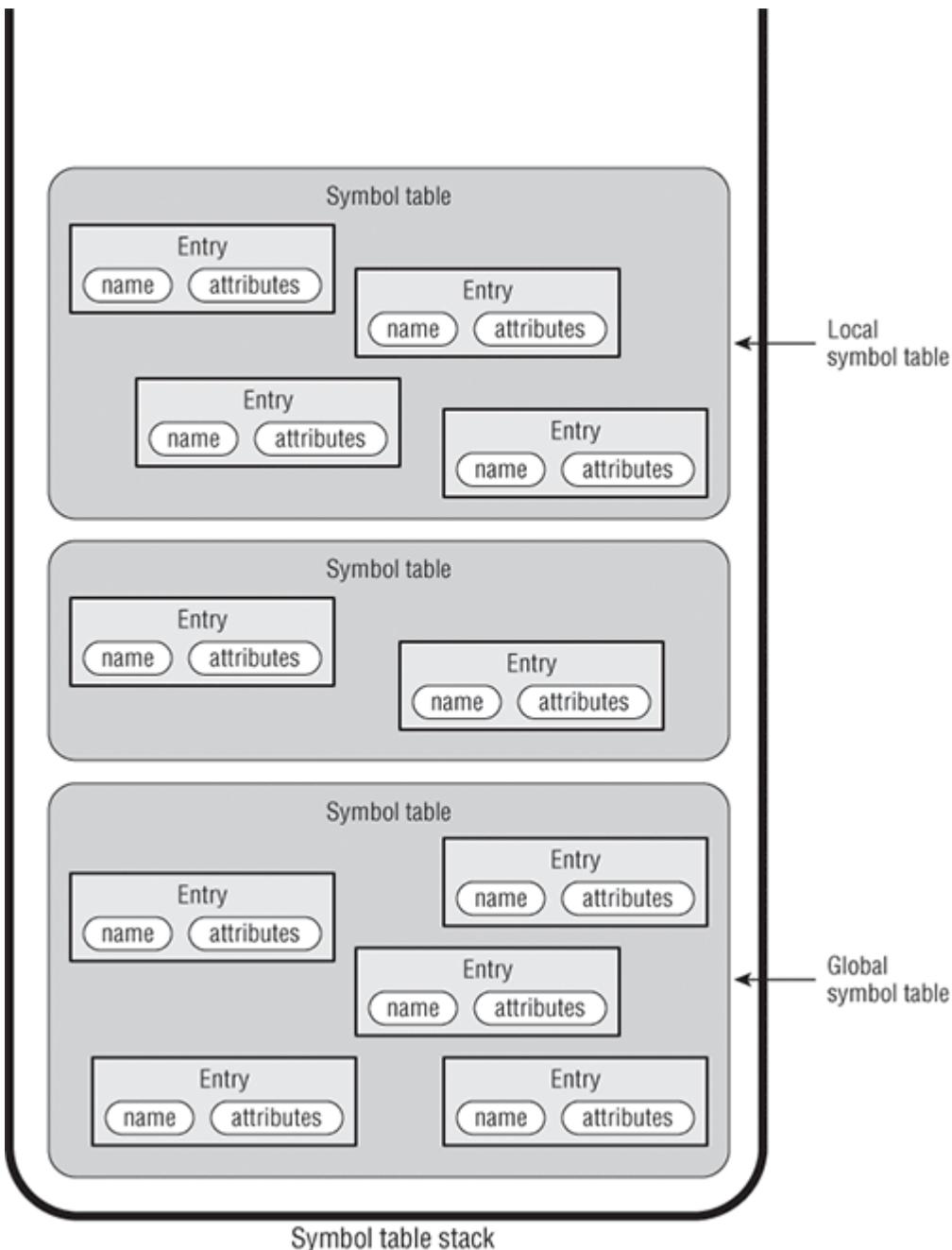
To parse a block-structured language such as Pascal, you'll actually need multiple symbol tables – a *global* symbol table: one for the main program, one for each procedure and function, and one for every record type. Because Pascal routines and records can be nested, these symbol tables need to be maintained on a stack. The symbol table at the top of the stack maintains information for the program, routine, or record that the parser is currently working on. As the parser works its way through a Pascal program and enters and leaves nested routines and record type definitions, it pushes and pops symbol tables from the stack. The symbol table currently at the top of the stack is known as the *local* table.[1]

[1] If the global symbol table is the only one on the stack, it is also the local table.

Figure 4-1 shows the conceptual design for the symbol table stack, the symbol tables, and symbol table entries. In the conceptual design, the symbol table stack contains one or more symbol tables each of which contains symbol table entries. Each symbol table entry contains information about one token, typically an identifier, and consists of the entry's name and the token information in the form of attributes. A symbol table searches its entries by using the entry names as search keys.

At this point, you need not know, nor should you care, what data structures you'll use to build the symbol table or how you'll store the entries. From the conceptual design, you only need to understand what the major symbol table components are, what their roles are, and how they relate to each other.

Figure 4-1: Conceptual design of the symbol table stack, the symbol tables, and the symbol table entries. The symbol table at the top of the stack is the local table, and the one at the bottom is the global table.
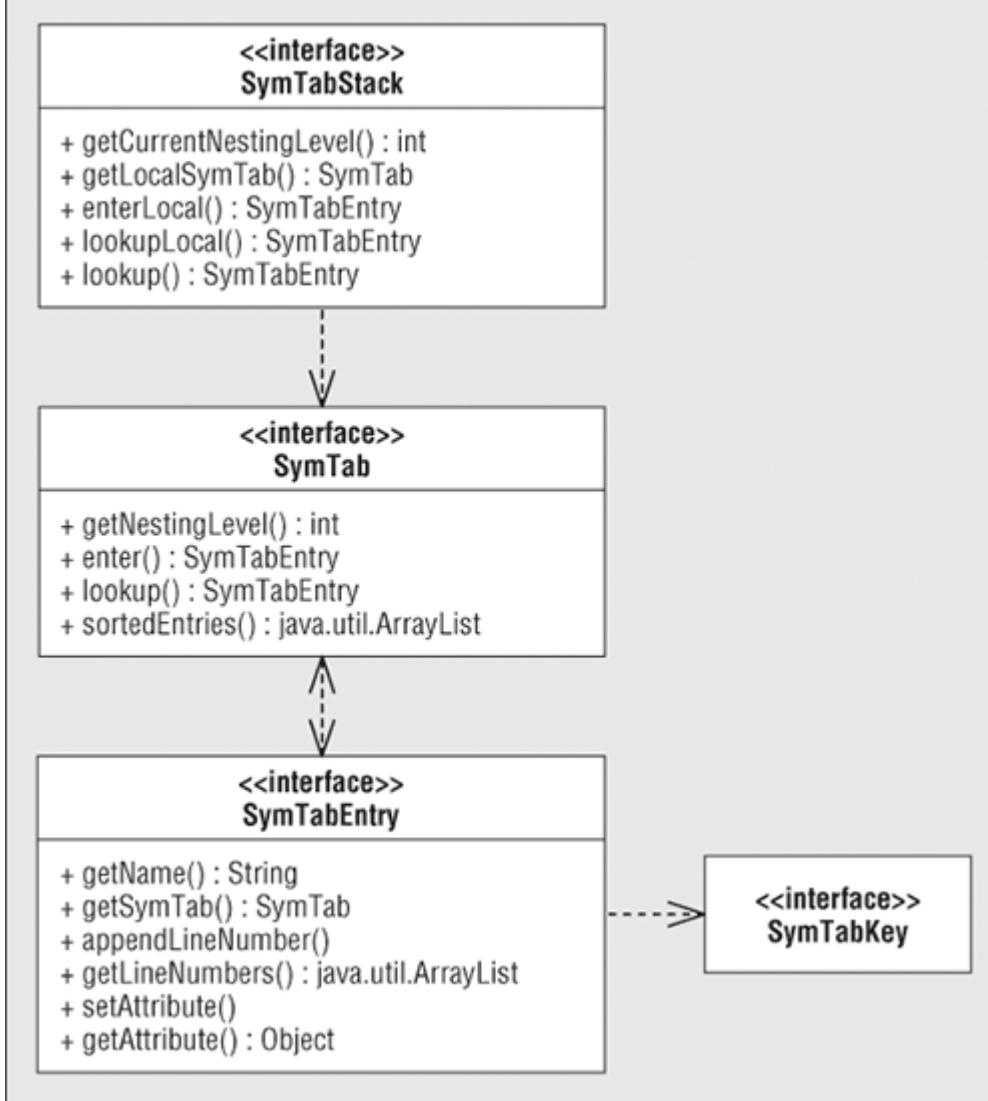
Symbol table stack

Design Note

Ideally, the other components of the compiler or interpreter will not need to know much about the symbol table beyond the conceptual level. This maintains loose coupling among the major components.

Until Chapter 9, you will only have a single symbol table in the stack. Defining the symbol table stack now makes it easier to make the transition to multiple symbol tables in the future.

Symbol Table Interfaces

From the symbol table components shown in
Figure 4-1, you can derive a UML diagram for the interfaces that represent them in the `intermediate` package. See Figure 4-2.

Figure 4-2: Symbol table interfaces.

Even though there is only a single table in the stack for now, you can introduce the key symbol table operations:

- **Enter** a new entry into the *local* symbol table, the table currently at the top of the stack.
- **Look up** an entry by searching only the *local* symbol table.
- **Look up** an entry by searching *all* the symbol tables in the stack.

Once a symbol table entry has been looked up (found), you can update its contents. Note that you can create a new entry only in the local table. However, you can look up an entry by searching either only the local table or all the tables in the stack.

Interface `SymTabStack()` supports these operations. Method `enterLocal()` creates a new entry in the local symbol table. Method `lookupLocal()` searches the local symbol table, and method `lookup()` searches all the tables in the stack. Interface `SymTabStack()` specifies two more methods: `getCurrentNestingLevel()` returns the current nesting level and `getLocalSymTab()` returns a reference to the local symbol table at the top of the stack. Don't be concerned about nesting levels until Chapter 9.

Interface `SymTab` represents a single symbol table. Methods `enter()` and `lookup()` enter and look up entries, respectively, in the table. Method `sortedEntries()` returns the entries in a table sorted by their names. Each symbol table also has a nesting level, which method `getNestingLevel()` returns. This method won't be meaningful until Chapter 9.

Interface `SymTabEntry` represents a single symbol table entry. Method `getName()` obtains the entry's name and methods `setAttribute()` and `getAttribute()` get and set entry information in the form of attributes. To support cross referencing, method `appendLineNumber()` stores the line number of a source line where the entry's name appears and method `getLineNumbers()` returns a list of these numbers. Each symbol table entry keeps a reference to the symbol table that contains it, and method `getSymTab()` returns this reference.

It is straightforward to create Java interfaces from the UML diagram in Figure 4-2. Listing 4-1 shows interface `SymTabStack`.

Listing 4-1: Interface `SymTabStack`

```
package wci.intermediate;

import java.util.ArrayList;

/**
 * <h1> SymTabStack</h1>
 *
 * <p>The interface for the symbol table stack.</p>
 */
public interface SymTabStack
{
    /**
     * Getter.
```

```
     * @return the current nesting level.
     */
    public int getCurrentNestingLevel();

    /**
     * Return the local symbol table which is at the top of the stack.
     * @return the local symbol table.
     */
    public SymTab getLocalSymTab();

    /**
     * Create and enter a new entry into the local symbol table.
     * @param name the name of the entry.
     * @return the new entry.
     */
    public SymTabEntry enterLocal(String name);

    /**
     * Look up an existing symbol table entry in the local symbol table.
     * @param name the name of the entry.
     * @return the entry, or null if it does not exist.
     */
    public SymTabEntry lookupLocal(String name);

    /**
     * Look up an existing symbol table entry throughout the stack.
     * @param name the name of the entry.
     * @return the entry, or null if it does not exist.
     */
    public SymTabEntry lookup(String name);
}
```

Chapter 2 contained an early version of interface `SymTab`. shows a more substantial version.

Listing 4-2: Interface `SymTab`

```
package wci.intermediate;

import java.util.ArrayList;

/**
 * <h1>SymTab</h1>
 *
 * <p>The framework interface that represents the symbol table.</p>
 */
public interface SymTab
{
    /**
     * Getter.
     * @return the scope nesting level of this entry.
     */
    public int getNestingLevel();

    /**
     * Create and enter a new entry into the symbol table.
     * @param name the name of the entry.
     * @return the new entry.
     */
    public SymTabEntry enter(String name);

    /**
     * Look up an existing symbol table entry.
     * @param name the name of the entry.
     * @return the entry, or null if it does not exist.
     */
    public SymTabEntry lookup(String name);

    /**
     * @return a list of symbol table entries sorted by name.
     */
    public ArrayList<SymTabEntry> sortedEntries();
}
```

shows interface `SymTabEntry`.

Listing 4-3: Interface `SymTabEntry`

```
package wci.intermediate;

import java.util.ArrayList;

/**
 * <h1>SymTabEntry</h1>
 *
 * <p>The interface for a symbol table entry.</p>
 */
public interface SymTabEntry
{
    /**
     * Getter.
     * @return the name of the entry.
     */
    public String getName();

    /**
     * Getter.
     * @return the symbol table that contains this entry.
     */
    public SymTab getSymTab();

    /**
     * Append a source line number to the entry.
     * @param lineNumber the line number to append.
     */
    public void appendLineNumber(int lineNumber);

    /**
     * Getter.
```

```
    * @return the list of source line numbers.
    */
   public ArrayList<Integer> getLineNumbers();


   /**
    * Set an attribute of the entry.
    * @param key the attribute key.
    * @param value the attribute value.
    */
   public void setAttribute(SymTabKey key, Object value);


   /**
    * Get the value of an attribute of the entry.
    * @param key the attribute key.
    * @return the attribute value.
    */
   public Object getAttribute(SymTabKey key);
}
```

 Finally, [Listing 4-4](#) shows the marker interface `SymTabKey`, which represents a key for accessing an entry's attribute information.

[Listing 4-4:](#) Interface `SymTabKey`
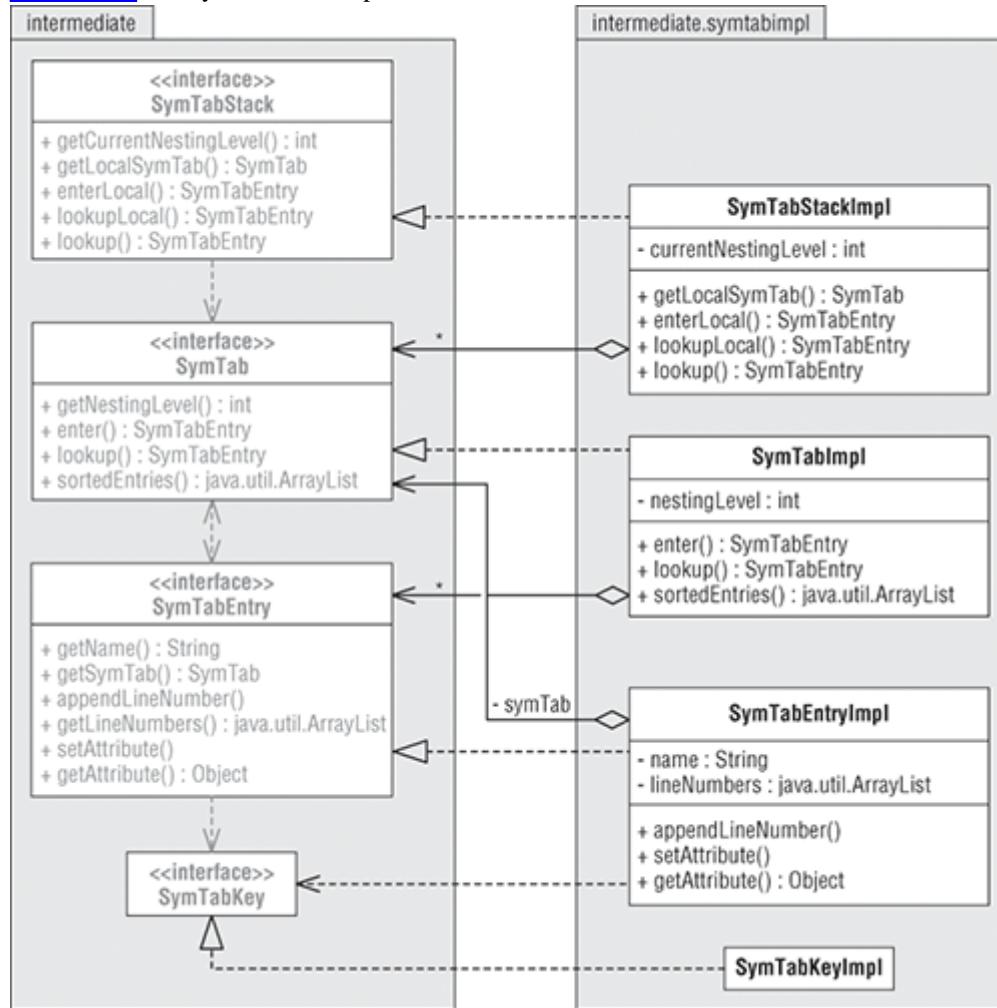
```
package wci.intermediate;

public interface SymTabKey
{
}
```

A Symbol Table Factory

[Figure 4-3](#) shows the UML diagram of the symbol table implementation classes, which are in package `intermediate.symtabimpl`.

[Figure 4-3:](#) The symbol table implementation classes

Design Note

The * at the arrowhead end of an ownership arrow indicates multiplicity. [Figure 4-3](#) shows that a `SymTabStackImpl` object can own 0 or more `SymTab` objects and that a `SymTab` object can own 0 or more `SymTabEntry` objects.

Design Note

Because you want the users of the symbol table to code only to its interfaces, you need a symbol table factory that isolates the users from the implementation classes. Listing 4-5 shows class `SymTabFactory`. Each method constructs an instance of one of the implementation classes.

Listing 4-5: Class `SymTabFactory`

```
package wci.intermediate;

import wci.intermediate.symtabimpl.*;

/**
 * <h1>SymTabFactory</h1>
 *
 * <p>A factory for creating objects that implement the symbol table.</p>
 */
public class SymTabFactory
{
    /**
     * Create and return a symbol table stack implementation.
     * @return the symbol table implementation.
     */
    public static SymTabStack createSymTabStack()
    {
        return new SymTabStackImpl();
    }

    /**
     * Create and return a symbol table implementation.
     * @param nestingLevel the nesting level.
     * @return the symbol table implementation.
     */
    public static SymTab createSymTab(int nestingLevel)
    {
        return new SymTabImpl(nestingLevel);
    }

    /**
     * Create and return a symbol table entry implementation.
     * @param name the identifier name.
     * @param symTab the symbol table that contains this entry.
     * @return the symbol table entry implementation.
     */
    public static SymTabEntry createSymTabEntry(String name, SymTab symTab)
    {
        return new SymTabEntryImpl(name, symTab);
    }
}
```

Because a compiler or an interpreter has only a single symbol table stack, you can refer to it with a static field. Listing 4-6 shows static field `symTabStack` in the framework `Parser` class.

Listing 4-6: Static fields of the framework `Parser` class

```
protected static SymTabStack symTabStack;       // symbol table stack
protected static MessageHandler messageHandler;  // message handler delegate

static {
    symTabStack = SymTabFactory.createSymTabStack();
    messageHandler = new MessageHandler();
}
```

Symbol Table Implementation

Now you're ready to develop the code that implements the symbol table interfaces.

Figure 4-3 shows that each class implements its interface and maintains an ownership relationship to the next *lower* interface. For example, class `SymTabImpl` implements interface `SymTab`, and each `SymTabImpl` object owns 0 or more `SymTabEntry` objects. Also, a `SymTabEntryImpl` object refers to the `SymTab` object that contains it.

Design Note

Because the implementation classes themselves code to the interfaces, none of the implementation classes depends on another implementation class.

Class `SymTabStackImpl` implements the interface `SymTabStack` and extends `java.util.ArrayList<SymTab>`. In other words, implement the symbol table stack as an array list. Listing 4-7 shows the key methods of the class. Note that the constructor creates and adds the first symbol table to the stack (the global table).

Listing 4-7: Key methods of class `SymTabStackImpl`

```
package wci.intermediate.symtabimpl;

import java.util.ArrayList;

import wci.intermediate.*;

/**
 * <h1>SymTabStack</h1>
 *
```

```
 * <p>An implementation of the symbol table stack.</p>
 */
public class SymTabStackImpl
    extends ArrayList<SymTab>
    implements SymTabStack
{
    private int currentNestingLevel;  // current scope nesting level

    /**
     * Constructor.
     */
    public SymTabStackImpl()
    {
        this.currentNestingLevel = 0;
        add(SymTabFactory.createSymTab(currentNestingLevel));
    }

    /**
     * Return the local symbol table which is at the top of the stack.
     * @return the local symbol table.
     */
    public SymTab getLocalSymTab()
    {
        return get(currentNestingLevel);
    }

    /**
     * Create and enter a new entry into the local symbol table.
     * @param name the name of the entry.
     * @return the new entry.
     */
    public SymTabEntry enterLocal(String name)
    {
        return get(currentNestingLevel).enter(name);
    }

    /**
     * Look up an existing symbol table entry in the local symbol table.
     * @param name the name of the entry.
     * @return the entry, or null if it does not exist.
     */
    public SymTabEntry lookupLocal(String name)
    {
        return get(currentNestingLevel).lookup(name);
    }

    /**
     * Look up an existing symbol table entry throughout the stack.
     * @param name the name of the entry.
     * @return the entry, or null if it does not exist.
     */
    public SymTabEntry lookup(String name)
    {
        return lookupLocal(name);
    }
}
```

Methods `enterLocal()` and `lookupLocal()` only involve the local symbol table at the top of the stack. For now, field `currentNestingLevel` will always be 0. Again for now, since there is only one symbol table on the stack, methods `lookup()` and `lookupLocal()` are functionally equivalent. Method `get()` is defined by the base `ArrayList` class.

Listing 4-8 shows the key methods of class `SymTabImpl`, which implements the interface `SymTab`. It extends `java.util.TreeMap`, which is a hash table that keeps its entries sorted in ascending order by key.

Listing 4-8: Key methods of class `SymTabImpl`

```
package wci.intermediate.symtabimpl;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.TreeMap;

import wci.intermediate.*;

/**
 * <h1>SymTabImpl</h1>
 *
 * <p>An implementation of the symbol table.</p>
 */
public class SymTabImpl
    extends TreeMap<String, SymTabEntry>
    implements SymTab
{
    private int nestingLevel;

    public SymTabImpl(int nestingLevel)
    {
        this.nestingLevel = nestingLevel;
    }

    /**
     * Create and enter a new entry into the symbol table.
     * @param name the name of the entry.
     * @return the new entry.
     */
    public SymTabEntry enter(String name)
    {
        SymTabEntry entry = SymTabFactory.createSymTabEntry(name, this);
        put(name, entry);

        return entry;
    }

    /**
     * Look up an existing symbol table entry.
     * @param name the name of the entry.
     * @return the entry, or null if it does not exist.
     */
    public SymTabEntry lookup(String name)
    {
        return get(name);
```

```
        }

        /**
         * @return a list of symbol table entries sorted by name.
         */
        public ArrayList<SymTabEntry> sortedEntries()
        {
            Collection<SymTabEntry> entries = values();
            Iterator<SymTabEntry> iter = entries.iterator();
            ArrayList<SymTabEntry> list = new ArrayList<SymTabEntry>(size());

            // Iterate over the sorted entries and append them to the list.
            while (iter.hasNext()) {
                list.add(iter.next());
            }

            return list;  // sorted list of entries
        }
    }
```

When each symbol table is created, its `nestingLevel` field is set to the current nesting level. For now, this will always be 0. Methods `enter()` and `lookup()` use the underlying hash table's `put()` and `get()` methods, respectively; `enter()` first calls the symbol table factory to create a `SymTabEntry` object with a given name. Method `sortedEntries()` uses the sorting capabilities of the `TreeMap` class to return a sorted list of entries.

Class `SymTabEntryImpl` implements the interface `SymTabEntry` and extends `java.util.HashMap`. Listing 4-9 shows its key methods. As you saw in the conceptual design in Figure 4-1, a symbol table entry consists of a name and a set of attributes. In Figure 4-3, you added fields `symTab` and `lineNumbers` to hold a references to the containing symbol table and to the array of source line numbers, respectively.

Listing 4-9: Key methods of class `SymTabEntryImpl`

```
package wci.intermediate.symtabimpl;

import java.util.ArrayList;
import java.util.HashMap;

import wci.intermediate.*;

/**
 * <h1>SymTabEntryImpl</h1>
 *
 * <p>An implementation of a symbol table entry.</p>
 */
public class SymTabEntryImpl
    extends HashMap<SymTabKey, Object>
    implements SymTabEntry
{
    private String name;                    // entry name
    private SymTab symTab;                   // parent symbol table
    private ArrayList<Integer> lineNumbers;  // source line numbers

    /**
     * Constructor.
     * @param name the name of the entry.
     * @param symTab the symbol table that contains this entry.
     */
    public SymTabEntryImpl(String name, SymTab symTab)
    {
        this.name = name;
        this.symTab = symTab;
        this.lineNumbers = new ArrayList<Integer>();
    }

    /**
     * Append a source line number to the entry.
     * @param lineNumber the line number to append.
     */
    public void appendLineNumber(int lineNumber)
    {
        lineNumbers.add(lineNumber);
    }

    /**
     * Set an attribute of the entry.
     * @param key the attribute key.
     * @param value the attribute value.
     */
    public void setAttribute(SymTabKey key, Object value)
    {
        put(key, value);
    }

    /**
     * Get the value of an attribute of the entry.
     * @param key the attribute key.
     * @return the attribute value.
     */
    public Object getAttribute(SymTabKey key)
    {
        return get(key);
    }
}
```

Methods `setAttribute()` and `getAttribute()` use the base hash table's `put()` and `get()` methods, respectively.

Design Note

Using a hash map to implement a symbol table entry provides the most flexibility in what information you can store in each entry.

Listing 4-10 shows the enumerated type `SymTabKeyImpl`, which implements interface `SymTabKey`. Subsequent chapters will use these keys.

Listing 4-10: **Enumerated type** `SymTabKeyImpl`

```
package wci.intermediate.symtabimpl;

import wci.intermediate.SymTabKey;

/**
 * <h1>SymTabKeyImpl</h1>
 *
 * <p>Attribute keys for a symbol table entry.</p>
 */
public enum SymTabKeyImpl implements SymTabKey
{
    // Constant.
    CONSTANT_VALUE,

    // Procedure or function.
    ROUTINE_CODE, ROUTINE_SYMTAB, ROUTINE_ICODE,
    ROUTINE_PARMS, ROUTINE_ROUTINES,

    // Variable or record field value.
    DATA_VALUE
}
```

   This completes the first goal of this chapter, creating a flexible, language-independent symbol table. The symbol table is flexible because users code only to its interfaces, thus allowing you to make future changes to its implementation. The symbol table entries can store any information. There are no Pascal-specific constraints.

Program 4: Pascal Cross-Referencer I

You're ready to verify the new symbol table. You can accomplish the goal by generating a cross-reference listing of the identifiers in a Pascal source program.

Listing 4-11 shows some sample output produced by a command line similar to

```
java -classpath classes Pascal compile -x newton.pas
```

The –x option generates the cross-reference listing.

Listing 4-11: A generated cross-reference listing

```
001 PROGRAM newton (input, output);
002
003 CONST
004     EPSILON = 1e-6;
005
006 VAR
007     number      : integer;
008     root, sqRoot : real;
009
010 BEGIN
011     REPEAT
012         writeln;
013         write('Enter new number (0 to quit): ');
014         read(number);
015
016         IF number = 0 THEN BEGIN
017             writeln(number:12, 0.0:12:6);
018         END
019         ELSE IF number < 0 THEN BEGIN
020             writeln('*** ERROR:  number < 0');
021         END
022         ELSE BEGIN
023             sqRoot := sqrt(number);
024             writeln(number:12, sqRoot:12:6);
025             writeln;
026
027             root := 1;
028             REPEAT
029                 root := (number/root + root)/2;
030                 writeln(root:24:6,
031                         100*abs(root - sqRoot)/sqRoot:12:2,
032                         '%')
033             UNTIL abs(number/sqr(root) - 1) < EPSILON;
034         END
035     UNTIL number = 0
036 END.

                36 source statements.
                 0 syntax errors.
               0.05 seconds total parsing time.


===== CROSS-REFERENCE TABLE =====

Identifier      Line numbers
----------      ------------
abs             031 033
epsilon         004 033
input           001
integer         007
newton          001
number          007 014 016 017 019 023 024 029 033 035
output          001
read            014
real            008
root            008 027 029 029 029 030 031 033
sqr             033
sqroot          008 023 024 031 031
sqrt            023
write           013
writeln         012 017 020 024 025 030

                 0 instructions generated.
```

After the source program listing, all of the source program's identifiers are listed alphabetically. Following each identifier name are the source line numbers where the identifier appears. Note that all the identifier names have been shifted to lower case.

Modify the `parse()` method of the parser subclass `PascalParserTD` to enter identifiers into the symbol table. See Listing 4-12.

Listing 4-12: Method `parse()` of class `PascalParserTD`

```
/**
 * Parse a Pascal source program and generate the symbol table
 * and the intermediate code.
 */
public void parse()
    throws Exception
{
    Token token;
    long startTime = System.currentTimeMillis();

    try {
        // Loop over each token until the end of file.
        while (!((token = nextToken()) instanceof EofToken)) {
            TokenType type = token.getType();

            // Cross reference only the identifiers.
            if (tokenType == IDENTIFIER) {
                String name = token.getText().toLowerCase();

                // If it&apos;s not already in the symbol table,
                // create and enter a new entry for the identifier.
                SymTabEntry entry = symTabStack.lookup(name);
                if (entry == null) {
                    entry = symTabStack.enterLocal(name);
                }

                // Append the current line number to the entry.
                entry.appendLineNumber(token.getLineNumber());
            }

            else if (tokenType == ERROR) {
                errorHandler.flag(token, (PascalErrorCode) token.getValue(),
                                  this);
            }
        }

        // Send the parser summary message.
        float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
        sendMessage(new Message(PARSER_SUMMARY,
                          new Number[] {token.getLineNumber(),
                                        getErrorCount(),
                                        elapsedTime}));
    }
    catch (java.io.IOException ex) {
        errorHandler.abortTranslation(IO_ERROR, this);
    }
}
```

The modified `parse()` method uses the symbol table factory to create a symbol table stack. It loops over all the tokens in the source program. For each token of type `IDENTIFIER`, it tries to look up the identifier in the local symbol table. If the identifier isn't already in there, the method enters it into the local symbol table. The method calls `entry.appendLineNumber()` to append the current source line number. Note that because Pascal is not case-sensitive, the call to the string method `toLowerCase()` "normalizes" the identifier name by shifting it to lower case:

```
String name = token.getText().toLowerCase();
```

Listing 4-13 shows the new utility class `CrossReferencer` in the `util` package.

Listing 4-13: Class `CrossReferencer`

```
package wci.util;

import java.util.ArrayList;

import wci.intermediate.*;
import static wci.intermediate.symtabimpl.SymTabKeyImpl.*;

/**
 * <h1>CrossReferencer</h1>
 *
 * <p>Generate a cross-reference listing.</p>
 *
 * <p>Copyright (c) 2008 by Ronald Mak</p>
 * <p>For instructional purposes only.  No warranties.</p>
 */
public class CrossReferencer
{
    private static final int NAME_WIDTH = 16;

    private static final String NAME_FORMAT       = "%-" + NAME_WIDTH + "s";
    private static final String NUMBERS_LABEL     = " Line numbers    ";
    private static final String NUMBERS_UNDERLINE = " ------------    ";
    private static final String NUMBER_FORMAT = " %03d";

    private static final int LABEL_WIDTH  = NUMBERS_LABEL.length();
    private static final int INDENT_WIDTH = NAME_WIDTH + LABEL_WIDTH;

    private static final StringBuilder INDENT = new StringBuilder(INDENT_WIDTH);
    static {
        for (int i = 0; i < INDENT_WIDTH; ++i) INDENT.append(" ");
    }

    /**
     * Print the cross-reference table.
     * @param symTabStack the symbol table stack.
     */
    public void print(SymTabStack symTabStack)
    {
        System.out.println("\n===== CROSS-REFERENCE TABLE =====");
        printColumnHeadings();

        printSymTab(symTabStack.getLocalSymTab());
    }

    /**
```

```
 * Print column headings.
 */
private void printColumnHeadings()
{
    System.out.println();
    System.out.println(String.format(NAME_FORMAT, "Identifier")
                     + NUMBERS_LABEL);
    System.out.println(String.format(NAME_FORMAT, "----------")
                     + NUMBERS_UNDERLINE);
}

/**
 * Print the entries in a symbol table.
 * @param SymTab the symbol table.
 */
private void printSymTab(SymTab symTab)
{
    // Loop over the sorted list of symbol table entries.
    ArrayList<SymTabEntry> sorted = symTab.sortedEntries();
    for (SymTabEntry entry : sorted) {
        ArrayList<Integer> lineNumbers = entry.getLineNumbers();

        // For each entry, print the identifier name
        // followed by the line numbers.
        System.out.print(String.format(NAME_FORMAT, entry.getName()));
        if (lineNumbers != null) {
            for (Integer lineNumber : lineNumbers) {
                System.out.print(String.format(NUMBER_FORMAT, lineNumber));
            }
        }
        System.out.println();
    }
}
}
```

Method `printSymTab()` loops over the sorted list of symbol table entries. For each entry, it prints the identifier name followed by the list of source-line numbers.

You'll write a more sophisticated version of class `CrossReferencer` in Chapter 9 after you've parsed Pascal declarations.

Reuse the main `Pascal` class, with a few modifications to its constructor. See Listing 4-14.

Listing 4-14: Modifications to the constructor of the main `Pascal` class

```
private SymTabStack symTabStack;  // symbol table stack

public Pascal(String operation, String filePath, String flags)
{
    ...
    symTabStack = parser.getSymTabStack();

    if (xref) {
        CrossReferencer crossReferencer = new CrossReferencer();
        crossReferencer.print(symTabStack);
    }

    backend.process(iCode, symTabStack);
    ...
}
```

Replace field `symTab` with field `symTabStack`. Instead of the symbol table, the constructor now gets the symbol table stack from the parser, which it passes to the back end. If the `-x` command-line argument is present, the constructor creates a new `CrossReferencer` object and calls `crossReferencer.print(symTabStack)` to generate the cross-reference listing.

Chapter 5

Parsing Expressions and Assignment Statements

In the previous chapter, you saw how the parser builds and maintains the symbol table on the symbol table stack during the translation process. The parser also performs the semantic actions of building and maintaining intermediate code that represents the source program. As first explained in Chapter 1, the front end generates intermediate code that is an abstract, "pre-digested" form of the source program that the back end can process efficiently. It is a critical part of the interface between the front and back ends, as shown in Figures 1-3 and 2-1.

Goals and Approach

In this chapter, you will concentrate mostly on parsing expressions, but also include the assignment statements that contain the expressions, and the compound statements that contain the assignment statements. The goals are

- Parsers in the front end for certain Pascal constructs: compound statements, assignment statements, and expressions.
- Flexible, language-independent intermediate code generated by the parsers to represent these constructs.

This chapter is just the beginning. In Chapter 6 you will execute the expressions and assignment statements using only the intermediate code and the symbol table.
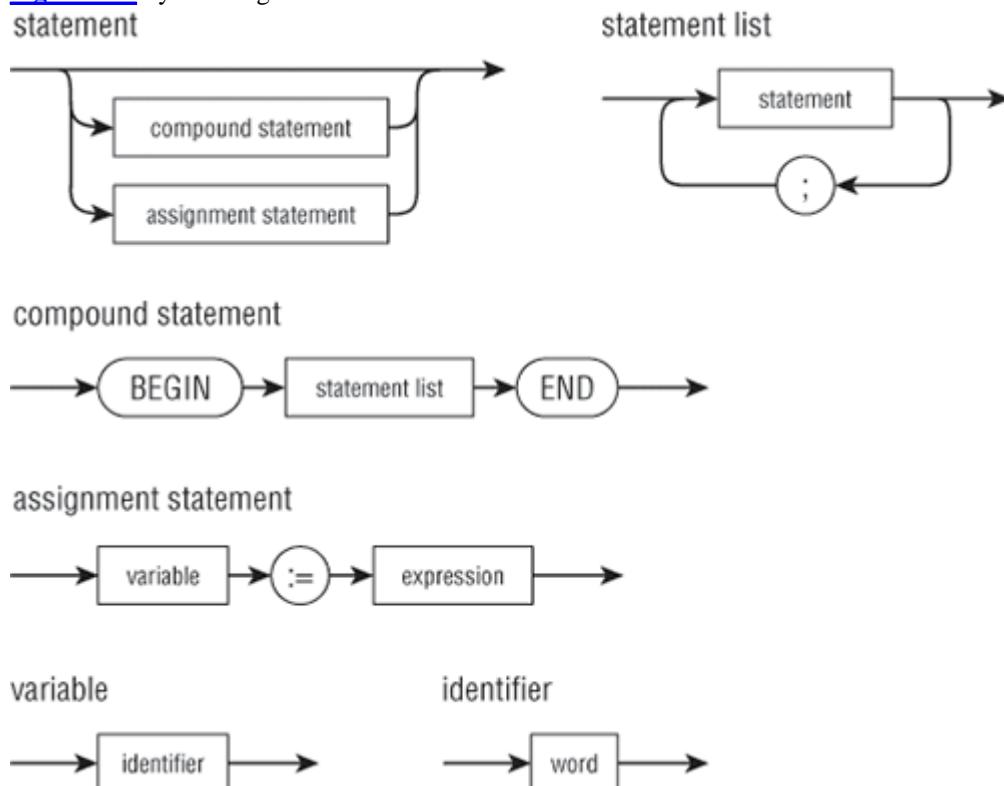
The approach followed in this chapter is to begin with syntax diagrams for the Pascal constructs. As you did in the previous chapter for the symbol table, you will create a conceptual design for the intermediate code structures, develop Java interfaces that represent the design, and write the Java classes that implement the interfaces. The syntax diagrams will guide the development of parsers that will generate the appropriate intermediate code. Finally, a syntax checker utility program will help verify the code you develop for this chapter.

Recall that parsing is also called *syntax analysis* and the parser is also known as the *syntax analyzer*.

Syntax Diagrams

Figure 5-1 shows syntax diagrams for a Pascal statement, statement list, compound statement, and assignment statement. Because a compound statement is itself a statement, compound statements can be nested. You will only handle compound and assignment statements (and the empty statement) in this chapter; later chapters will cover the rest of Pascal's statements.

Figure 5-1: Syntax diagrams for some Pascal statements
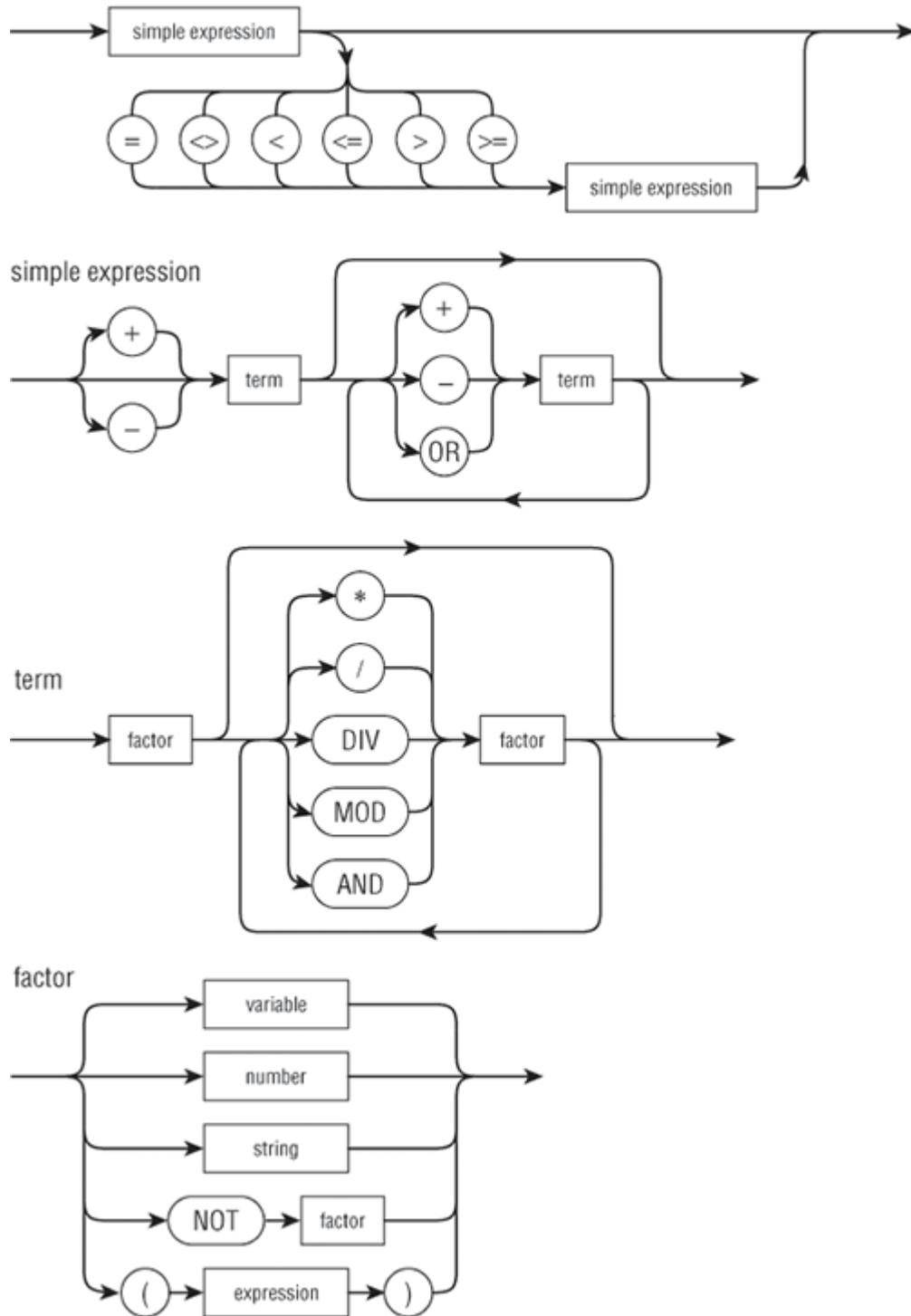
Figure 5-2 shows syntax diagrams for Pascal expressions.

The diagrams specify that an expression is composed of a single simple expression, or two simple expressions separated by a relational operator such as = or <=.[1] A simple expression is composed of one or more terms separated by an additive operator. A plus or minus sign may proceed the first term. A term is composed of one or more factors separated by operators. A factor is a variable, a number, a string, NOT followed by another factor, or a set of matched parentheses surrounding another expression. For now, a variable is simply an identifier, and an identifier is a word. Later chapters will handle variables that have subscripts and record fields.

[1] Note that Pascal uses = for the equality relational operator and := for the assignment operator, whereas Java uses == and =, respectively, for these operators.

Figure 5-2: Syntax diagrams for Pascal expressions

Design Note

Recall that rounded boxes in syntax diagrams represent terminal symbols and therefore each one surrounds literal text. Pascal has one- and two-character special symbols such as = and := and reserved words are not case sensitive.

The syntax diagrams in Figure 5-2 not only define a Pascal expression and its components recursively, but their hierarchical composition also specifies Pascal's operator precedence rules. If you start from the top diagram and work your way down to the bottom and consider each rectangular box in a diagram, such as `simple expression` and `term`, to be a "procedure call" to another

diagram, it becomes clear that operators in the lower diagrams bind tighter and execute before operators in the higher diagrams. Therefore, reading the diagrams from the bottommost upwards, Pascal has four levels of operator precedence:

Level	Operators
1 (highest)	NOT
2	multiplicative: `* / DIV MOD AND`
3	additive: `+ - OR`
4 (lowest)	relational: `= <> < <= > >=`
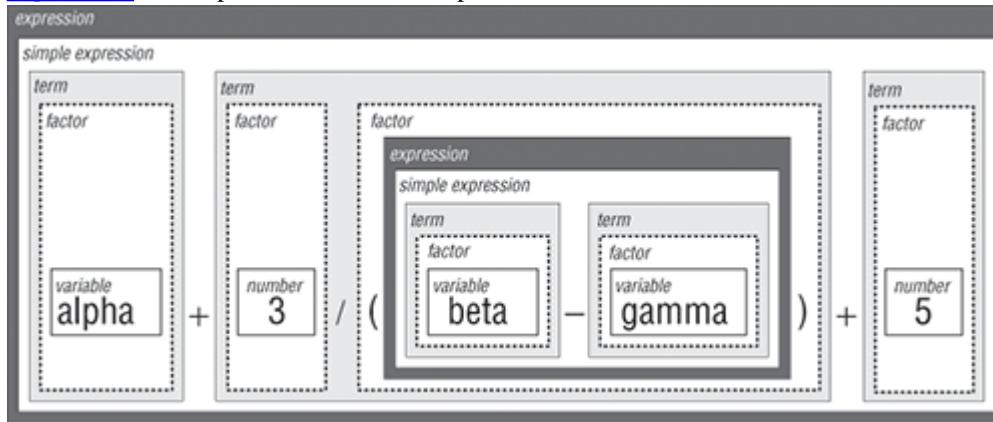
If there are no parentheses, higher level operators execute before the lower level ones, and operators at the same level execute from left to right. Because the bottommost diagram, `factor`, defines parenthesized expressions, parenthesized expressions always execute first, from the most deeply nested outwards. Within a parenthesized expression, the same precedence rules apply.

Figure 5-3 shows how to decompose the arithmetic expression

```
alpha + 3/(beta - gamma) + 5
```

according to the expression syntax diagrams.

Figure 5-3: Decomposition of a Pascal expression

Intermediate Code Conceptual Design
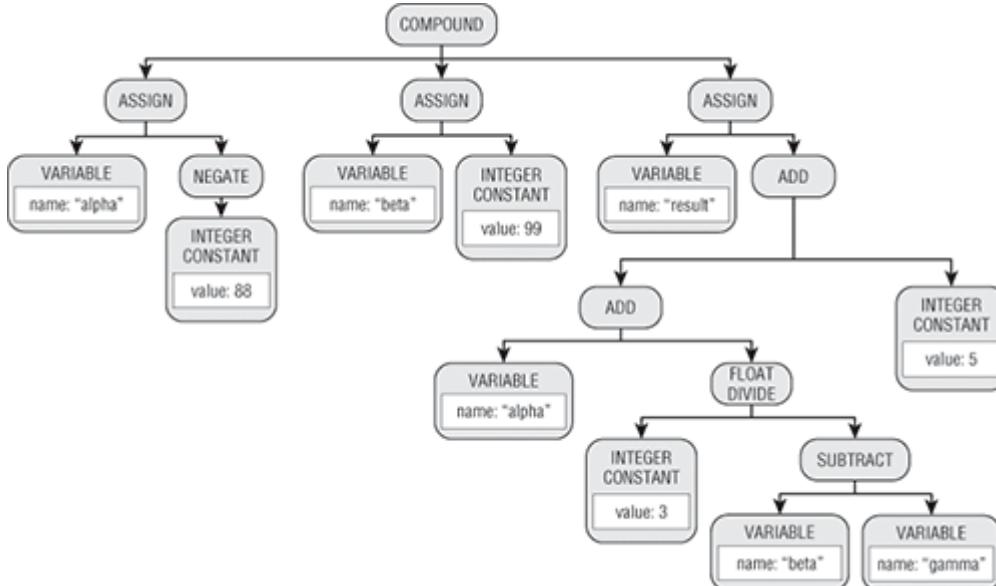
There are many ways to represent the intermediate code. You will use a tree data structure, and so the intermediate code takes the form of a *parse tree*.[2]

[2] This is more accurately called an *abstract syntax tree* or *AST* since certain source program tokens such as parentheses have been "abstracted out." But this book will continue to use the generic term *parse tree*.

Figure 5-4 shows the parse tree for the following compound statement:

```
BEGIN
    alpha  := -88;
    beta   := 99;
    result := alpha + 3/(beta - gamma) + 5
END
```

Figure 5-4: Intermediate code in the form of a parse tree

The conceptual design is straightforward. A parse tree consists of subtrees that represent Pascal constructs, such as statements and expressions. Each tree node has a node type and a set of attributes. For example, in Figure 5-4, one of the VARIABLE nodes has a name

attribute with the value `"alpha"`. Each node other than the root node has a single parent node. Different types of nodes can each have a certain number of child nodes. An `INTEGER_CONSTANT` node is a leaf node with no children. A `NEGATE` node has exactly one child. `ASSIGN` and `ADD` nodes have exactly two children each. A `COMPOUND` node can each have any number of children or none at all. A child can be a single node or the root node of a subtree.

The conceptual design doesn't say *how* you should implement a parse tree. But it's safe to assume that the basic tree operations that any implementation must support include:
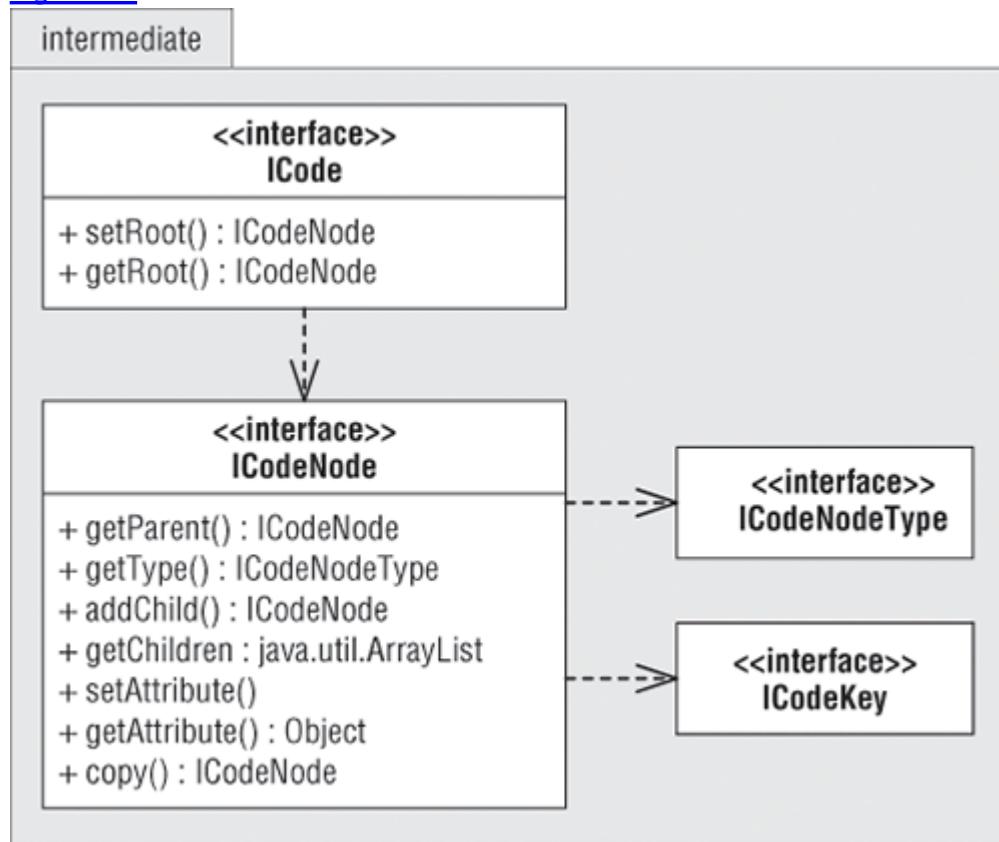
- Create a new node.
- Create a copy of a node.
- Set and get the root node of a parse tree.
- Set and get an attribute value in a node.
- Add a child node to a node.
- Get the list of a node's child nodes.
- Get a node's parent node.

Intermediate Code Interfaces

Just as you did for the symbol table in the previous chapter, first create intermediate code interfaces in the `intermediate` package, based on the conceptual design.
Figure 5-5 shows the UML diagram.

Figure 5-5: Intermediate code interfaces

Listing 5-1 shows interface `ICode`, which is an expanded version of the interface from Chapter 2.

Listing 5-1: Interface `ICode`

```
package wci.intermediate;

/**
 * <h1>ICode</h1>
 *
 * <p>The framework interface that represents the intermediate code.</p>
 */
public interface ICode
{
    /**
     * Set and return the root node.
     * @param node the node to set as root.
     * @return the root node.
     */
    public ICodeNode setRoot(ICodeNode node);

    /**
     * Get the root node.
     * @return the root node.
     */
    public ICodeNode getRoot();
}
```

<u>Listing 5-2</u> shows interface `ICodeNode`.

<u>Listing 5-2:</u> Interface `ICodeNode`

```
package wci.intermediate;

import java.util.ArrayList;

/**
 * <h1>ICodeNode</h1>
 *
 * <p>The interface for a node of the intermediate code.</p>
 */
public interface ICodeNode
{
    /**
     * Getter.
     * @return the node type.
     */
    public ICodeNodeType getType();

    /**
     * Return the parent of this node.
     * @return the parent node.
     */
    public ICodeNode getParent();

    /**
     * Add a child node.
     * @param node the child node.
     * @return the child node.
     */
    public ICodeNode addChild(ICodeNode node);

    /**
     * Return an array list of this node&apos;s children.
     * @return the array list of children.
     */
    public ArrayList<ICodeNode> getChildren();

    /**
     * Set a node attribute.
     * @param key the attribute key.
     * @param value the attribute value.
     */
    public void setAttribute(ICodeKey key, Object value);

    /**
     * Get the value of a node attribute.
     * @param key the attribute key.
     * @return the attribute value.
     */
    public Object getAttribute(ICodeKey key);

    /**
     * Make a copy of this node.
     * @return the copy.
     */
    public ICodeNode copy();
}
```

Any implementation of the methods defined in these two interfaces will perform the basic tree operations listed above.

<u>Listing 5-3</u> shows the marker interface `ICodeNodeType`, which represents the parse tree node types.

<u>Listing 5-3:</u> Interface `ICodeNodeType`

```
package wci.intermediate;

public interface ICodeNodeType
{
}
```

<u>Listing 5-4</u> shows the marker interface `ICodeKey`, which represents the attribute keys of each parse tree node.

<u>Listing 5-4:</u> Interface `ICodeKey`

```
package wci.intermediate;

public interface ICodeKey
{
}
```

An Intermediate Code Factory

So far, the conceptual design and the interfaces do not rely on any particular implementation of the intermediate code. Any classes that use the intermediate code should program only to the interfaces in order to remain loosely coupled from the implementation.[3]

[3] Review the important design concepts of *loose coupling* and *coding to the interfaces* in the previous chapter.

Just as the symbol table needed a factory class to generate its components, so does the intermediate code.

<u>Listing 5-5</u> shows class `ICodeFactory`.

<u>Listing 5-5:</u> Class `ICodeFactory`

```
package wci.intermediate;

import wci.intermediate.icodeimpl.ICodeImpl;
import wci.intermediate.icodeimpl.ICodeNodeImpl;
```

```
/**
 * <h1>ICodeFactory</h1>
 *
 * <p>A factory for creating objects that implement the intermediate code.</p>
 */
public class ICodeFactory
{
    /**
     * Create and return an intermediate code implementation.
     * @return the intermediate code implementation.
     */
    public static ICode createICode()
    {
        return new ICodeImpl();
    }

    /**
     * Create and return a node implementation.
     * @param type the node type.
     * @return the node implementation.
     */
    public static ICodeNode createICodeNode(ICodeNodeType type)
    {
        return new ICodeNodeImpl(type);
    }
}
```

Intermediate Code Implementation

<u>Figure 5-6</u> shows the UML diagram for the classes in package `intermediate.icodeimpl` that implement the intermediate code interfaces. Class `ICodeImpl` implements interface `ICode` and class `ICodeNodeImpl` implements interface `ICodeNode`. Also, class `ICodeNodeTypeImpl` implements interface `ICodeNodeType` and class `ICodeKeyImpl` implements interface `ICodeKey`.

<u>Listing 5-6</u> shows the constructor of class `ICodeImpl`.

<u>Listing 5-6:</u> The constructor of class `ICodeImpl`
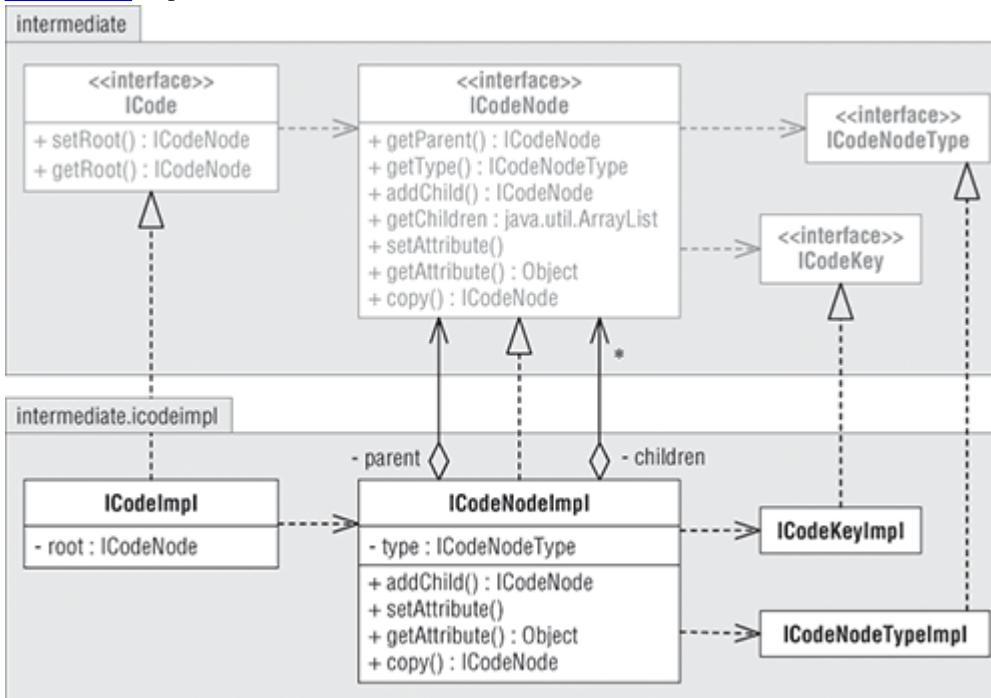
```
package wci.intermediate.icodeimpl;

import wci.intermediate.ICode;
import wci.intermediate.ICodeNode;

/**
 * <h1>ICodeImpl</h1>
 *
 * <p>An implementation of the intermediate code as a parse tree.</p>
 */
public class ICodeImpl implements ICode
{
    private ICodeNode root;  // root node

    /**
     * Set and return the root node.
     * @param node the node to set as root.
     * @return the root node.
     */
    public ICodeNode setRoot(ICodeNode node)
    {
        root = node;
        return root;
    }
}
```

<u>Figure 5-6:</u> Implementation classes for the intermediate code

<u>Listing 5-7</u> shows the key methods of class `ICodeNodeImpl`.

```
package wci.intermediate.icodeimpl;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

import wci.intermediate.*;

/**
 * <h1>ICodeNodeImpl</h1>
 *
 * <p>An implementation of a node of the intermediate code.</p>
 */
public class ICodeNodeImpl
    extends HashMap<ICodeKey, Object>
    implements ICodeNode
{
    private ICodeNodeType type;        // node type
    private ICodeNode parent;          // parent node
    private ArrayList<ICodeNode> children;  // children array list

    /**
     * Constructor.
     * @param type the node type whose name will be the name of this node.
     */
    public ICodeNodeImpl(ICodeNodeType type)
    {
        this.type = type;
        this.parent = null;
        this.children = new ArrayList<ICodeNode>();
    }

    /**
     * Add a child node.
     * @param node the child node. Not added if null.
     * @return the child node.
     */
    public ICodeNode addChild(ICodeNode node)
    {
        if (node != null) {
            children.add(node);
            ((ICodeNodeImpl) node).parent = this;
        }

        return node;
    }

    /**
     * Set a node attribute.
     * @param key the attribute key.
     * @param value the attribute value.
     */
    public void setAttribute(ICodeKey key, Object value)
    {
        put(key, value);
    }

    /**
     * Get the value of a node attribute.
     * @param key the attribute key.
     * @return the attribute value.
     */
    public Object getAttribute(ICodeKey key)
    {
        return get(key);
    }

    /**
     * Make a copy of this node.
     * @return the copy.
     */
    public ICodeNode copy()
    {
        // Create a copy with the same type.
        ICodeNodeImpl copy =
            (ICodeNodeImpl) ICodeFactory.createICodeNode(type);

        Set<Map.Entry<ICodeKey, Object>> attributes = entrySet();
        Iterator<Map.Entry<ICodeKey, Object>> it = attributes.iterator();

        // Copy attributes
        while (it.hasNext()) {
            Map.Entry<ICodeKey, Object> attribute = it.next();
            copy.put(attribute.getKey(), attribute.getValue());
        }

        return copy;
    }

    public String toString()
    {
        return type.toString();
    }
}
```

This is a very straightforward implementation of a parse tree node. The constructor creates a node with a given node type. The `copy()` method makes a copy of the node with the same node type. It copies all of the attributes to the new node.

Design Note

Class `ICodeNodeImpl` extends `java.util.HashMap`. Similarly to how you implemented a symbol table entry, implementing a parse tree node as a hash table provides the most flexibility in what attributes you can store in each node.

Listing 5-8 shows the enumerated type `ICodeNodeTypeImpl`. Its enumerated values represent all the parse tree node types.

```
package wci.intermediate.icodeimpl;

import wci.intermediate.ICodeNodeType;

/**
 * <h1>ICodeNodeType</h1>
 *
 * <p>Node types of the intermediate code parse tree.</p>
 */
public enum ICodeNodeTypeImpl implements ICodeNodeType
{
    // Program structure
    PROGRAM, PROCEDURE, FUNCTION,

    // Statements
    COMPOUND, ASSIGN, LOOP, TEST, CALL, PARAMETERS,
    IF, SELECT, SELECT_BRANCH, SELECT_CONSTANTS, NO_OP,

    // Relational operators
    EQ, NE, LT, LE, GT, GE, NOT,

    // Additive operators
    ADD, SUBTRACT, OR, NEGATE,

    // Multiplicative operators
    MULTIPLY, INTEGER_DIVIDE, FLOAT_DIVIDE, MOD, AND,

    // Operands
    VARIABLE, SUBSCRIPTS, FIELD,
    INTEGER_CONSTANT, REAL_CONSTANT,
    STRING_CONSTANT, BOOLEAN_CONSTANT,
}
```

Design Note

By defining a separate set of enumerated values to represent the parse tree node types, instead of using values from the enumerated type `PascalTokenType`, the intermediate code can remain language independent.

[Listing 5-9](#) shows the enumerated type `ICodeKeyImpl`.

[Listing 5-9:](#) Enumerated type `ICodeKeyImpl`

```
package wci.intermediate.icodeimpl;

import wci.intermediate.ICodeKey;

/**
 * <h1>ICodeKeyImpl</h1>
 *
 * <p>Attribute keys for an intermediate code node.</p>
 */
public enum ICodeKeyImpl implements ICodeKey
{
    LINE, ID, VALUE;
}
```

You won't be storing many attribute values in the parse tree nodes. Most of the information will be encoded in the node type itself and in the tree structure.

Printing Parse Trees

The `-i` command line argument of the Pascal main program is a request to print the intermediate code, a useful feature for debugging. Since your intermediate code is a parse tree, you need a good format to print a tree.

The industry-standard XML can represent the tree structures in text form. For example, [Listing 5-10](#) shows the representation of the parse tree in [Figure 5-4](#), assuming the corresponding source statements have line numbers 18 through 22.

[Listing 5-10:](#) XML representation of the parse tree in Figure 5-4

```
<COMPOUND line="18">
    <ASSIGN line="19">
        <VARIABLE id="alpha" level="0" />
        <NEGATE>
            <INTEGER_CONSTANT value="88" />
        </NEGATE>
    </ASSIGN>
    <ASSIGN line="20">
        <VARIABLE id="beta" level="0" />
        <INTEGER_CONSTANT value="99" />
    </ASSIGN>
    <ASSIGN line="21">
        <VARIABLE id="result" level="0" />
        <ADD>
            <ADD>
                <VARIABLE id="alpha" level="0" />
                <FLOAT_DIVIDE>
                    <INTEGER_CONSTANT value="3" />
                    <SUBTRACT>
                        <VARIABLE id="beta" level="0" />
                        <VARIABLE id="gamma" level="0" />
                    </SUBTRACT>
                </FLOAT_DIVIDE>
            </ADD>
            <INTEGER_CONSTANT value="5" />
        </ADD>
    </ASSIGN>
</COMPOUND>
```

<div style="border:1px solid #999; padding:1em; background:#f0f0f0;">

Design Note

The Extensible Markup Language (XML) is an industry standard for representing structured data. Its popularity has grown with the Internet, and it is a common way to transport data across the network, especially with web services. There are now different forms of XML for specific application domains and numerous APIs, tools, and utilities to generate and manipulate XML.

An XML document consists of elements. An element is represented by a tag, such as <address>. An element can contain content, either text or child elements. If it has content, the element surrounds the content with opening and closing tags. For example:

```
<address>

<street>123 Main Street</street>

<city>San Jose</city>

<state>CA</state>

</address>
```

The matching closing tag has the same name as the opening tag but contains an initial /, such as `</address>`. An element that has no content can be written either as

```
<homeowner></homeowner>
```

or more simply as

```
<homeowner />
```

An element can also have one or more attributes represented by name-value pairs. Attributes are written inside the element's opening tag with each value in quotes:

```
<description units="metric">
    <size length="5" width="9" />
    <velocity speed="8" heading="11,25,7" />
</description>
```

When written as text, XML is free-form. XML is easier (for humans) to read when each element tag begins on a new line and child tags are indented with respect to their parent tags, as shown above.

There is much more to XML than this, but this brief description should be all that you'll need for this book.

</div>

shows class `ParseTreePrinter` in the `util` utilities package. It prints an XML representation of a parse tree, such as the one in .

Listing 5-11: Class `ParseTreePrinter`

```java
package wci.util;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.io.PrintStream;

import wci.intermediate.*;
import wci.intermediate.icodeimpl.*;

/**
 * <h1>ParseTreePrinter</h1>
 *
 * <p>Print a parse tree.</p>
 */
public class ParseTreePrinter
{
    private static final int INDENT_WIDTH = 4;
    private static final int LINE_WIDTH = 80;

    private PrintStream ps;      // output print stream
    private int length;          // output line length
    private String indent;       // indent spaces
    private String indentation;  // indentation of a line
    private StringBuilder line;  // output line

    /**
     * Constructor
     * @param ps the output print stream.
     */
    public ParseTreePrinter(PrintStream ps)
    {
        this.ps = ps;
        this.length = 0;
        this.indentation = "";
        this.line = new StringBuilder();

        // The indent is INDENT_WIDTH spaces.
        this.indent = "";
        for (int i = 0; i < INDENT_WIDTH; ++i) {
            this.indent += " ";
        }
    }

    /**
     * Print the intermediate code as a parse tree.
     * @param iCode the intermediate code.
     */
    public void print(ICode iCode)
    {
        ps.println("\n===== INTERMEDIATE CODE =====\n");

        printNode((ICodeNodeImpl) iCode.getRoot());
        printLine();
    }

    /**
     * Print a parse tree node.
     * @param node the parse tree node.
     */
    private void printNode(ICodeNodeImpl node)
    {
        // Opening tag.
        append(indentation); append("<" + node.toString());

        printAttributes(node);
        printTypeSpec(node);
```

```java
        ArrayList<ICodeNode> childNodes = node.getChildren();

        // Print the node&apos;s children followed by the closing tag.
        if ((childNodes != null) && (childNodes.size() > 0)) {
            append(">");
            printLine();

            printChildNodes(childNodes);
            append(indentation); append("</" + node.toString() + ">");
        }

        // No children: Close off the tag.
        else {
            append(" "); append("/>");
        }

        printLine();
    }

    /**
     * Print a parse tree node&apos;s attributes.
     * @param node the parse tree node.
     */
    private void printAttributes(ICodeNodeImpl node)
    {
        String saveIndentation = indentation;
        indentation += indent;

        Set<Map.Entry<ICodeKey, Object>> attributes = node.entrySet();
        Iterator<Map.Entry<ICodeKey, Object>> it = attributes.iterator();

        // Iterate to print each attribute.
        while (it.hasNext()) {
            Map.Entry<ICodeKey, Object> attribute = it.next();
            printAttribute(attribute.getKey().toString(), attribute.getValue());
        }

        indentation = saveIndentation;
    }

    /**
     * Print a node attribute as key="value".
     * @param keyString the key string.
     * @param value the value.
     */
    private void printAttribute(String keyString, Object value)
    {
        // If the value is a symbol table entry, use the identifier&apos;s name.
        // Else just use the value string.
        boolean isSymTabEntry = value instanceof SymTabEntry;
        String valueString = isSymTabEntry ? ((SymTabEntry) value).getName()
                                           : value.toString();

        String text = keyString.toLowerCase() + "=\"" + valueString + "\"";
        append(" "); append(text);

        // Include an identifier&apos;s nesting level.
        if (isSymTabEntry) {
            int level = ((SymTabEntry) value).getSymTab().getNestingLevel();
            printAttribute("LEVEL", level);
        }
    }

    /**
     * Print a parse tree node&apos;s child nodes.
     * @param childNodes the array list of child nodes.
     */
    private void printChildNodes(ArrayList<ICodeNode> childNodes)
    {
        String saveIndentation = indentation;
        indentation += indent;

        for (ICodeNode child : childNodes) {
            printNode((ICodeNodeImpl) child);
        }

        indentation = saveIndentation;
    }

    /**
     * Print a parse tree node&apos;s type specification.
     * @param node the parse tree node.
     */
    private void printTypeSpec(ICodeNodeImpl node)
    {
    }

    /**
     * Append text to the output line.
     * @param text the text to append.
     */
    private void append(String text)
    {
        int textLength = text.length();
        boolean lineBreak = false;

        // Wrap lines that are too long.
        if (length + textLength > LINE_WIDTH) {
            printLine();
            line.append(indentation);
            length = indentation.length();
            lineBreak = true;
        }

        // Append the text.
        if (!(lineBreak && text.equals(" "))) {
            line.append(text);
            length += textLength;
        }
    }

    /**
     * Print an output line.
     */
    private void printLine()
    {
        if (length > 0) {
            ps.println(line);
            line.setLength(0);
            length = 0;
        }
```

```
        }
    }
```

The constructor expects a `PrintStream` argument such as `System.out`. The printing methods attempt to indent and wrap the printed XML lines properly, but are otherwise straightforward. They develop each print line in field `line`.

Method `printAttribute()` checks if the attribute value is a symbol table entry. If so, it prints the entry's name as the value. For a symbol table entry, it also prints the nesting level as the value of the artificial `LEVEL` attribute.

Method `printTypeSpec()` is currently only a placeholder. You'll complete this method in Chapter 10.

Parsing Pascal Statements and Expressions

Now you're ready to parse Pascal compound statements, assignment statements, and expressions and generate intermediate code in the form of a parse tree. In the next chapter, executors in the back end will interpret the parse tree.

Figure 5-7 shows the UML diagram of the Pascal parser subclass `StatementParser` in package `frontend.pascal.parsers`, which in turn is the superclass of `CompoundStatementParser`, `AssignmentStatementParser`, and `ExpressionParser`.

Design Note

Just as you developed separate subclasses of class `PascalToken` in Chapter 3 to represent each token type, now develop separate subclasses of class `StatementParser` to parse each Pascal construct. This adheres to the principle of designing highly cohesive, loosely coupled classes.
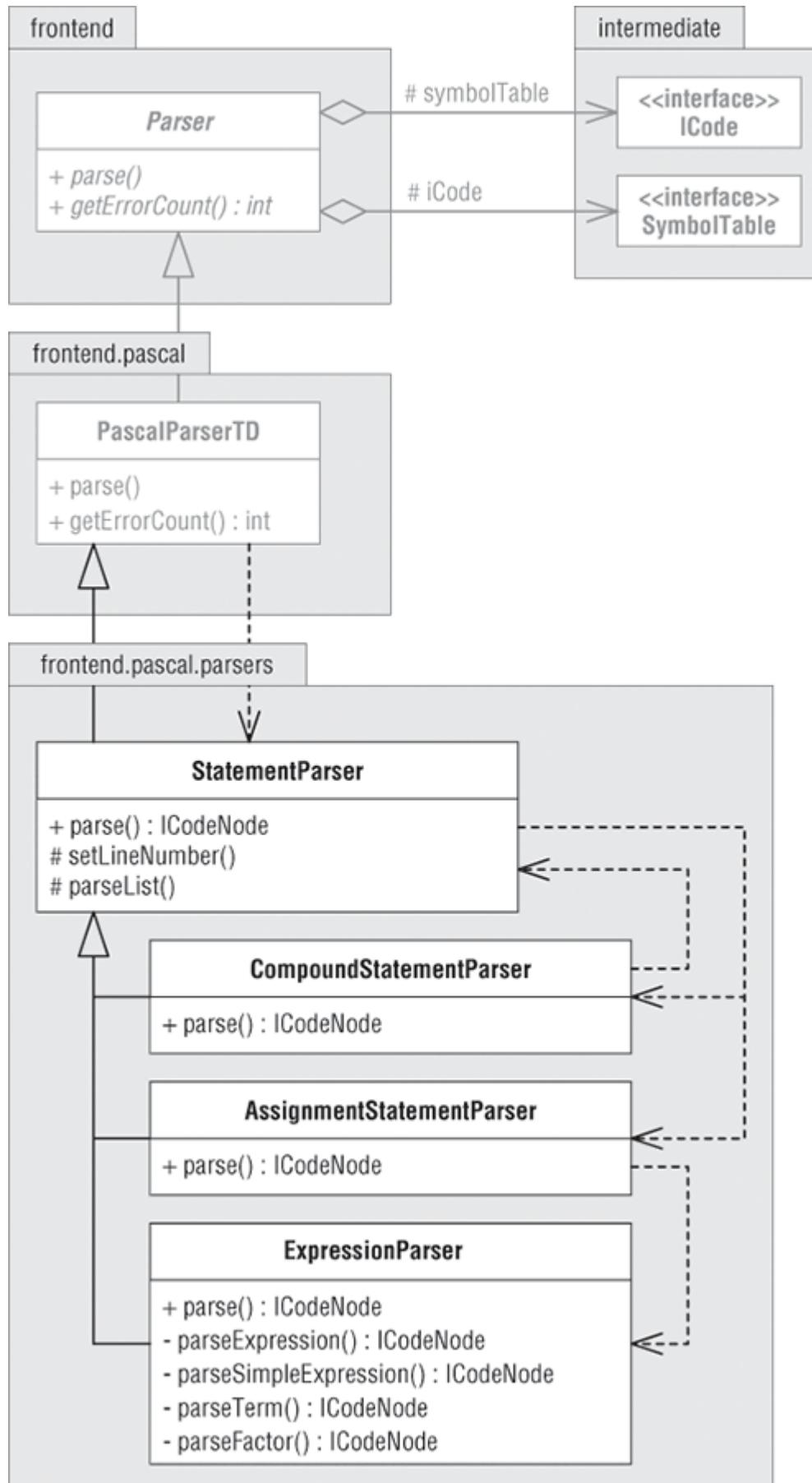
`StatementParser` depends on `CompoundStatementParser` and `AssignmentStatementParser`. Because a compound statement can contain nested statements, `CompoundStatementParser` depends on `StatementParser`. An assignment statement contains an expression, and so `AssignmentStatementParser` depends on `ExpressionParser`.

Class `StatementParser` and each of its subclasses has a `parse()` method that is specialized to parse a particular Pascal construct and generate the parse subtree for that construct. The method returns the `ICodeNode` root node of the generated subtree.

Class `StatementParser` also includes method `setLineNumber()` to set a statement node's source line number and method `parseList()` to parse statement lists. Class `ExpressionParser` has private parsing methods `parseExpression()`, `parseSimpleExpression()`, `parseTerm()`, and `parseFactor()` whose names mirror the names of the syntax diagrams in Figure 5-2.

Listing 5-12 shows file `assignments.txt` which will be our test source file. The source consists of an outer compound statement and ends with a period. Nested inside the outer compound statement are other compound and assignment statements. The nested compound statements contain still more assignment statements. You will parse this source file and generate the intermediate code.

Figure 5-7: Statement parser class `StatementParser` and its subclasses

frontend

Parser

+ parse()
+ getErrorCount() : int

symbolTable

iCode

intermediate

<<interface>>
ICode

<<interface>>
SymbolTable

frontend.pascal

PascalParserTD

+ parse()
+ getErrorCount() : int

frontend.pascal.parsers

StatementParser

+ parse() : ICodeNode
# setLineNumber()
# parseList()

CompoundStatementParser

+ parse() : ICodeNode

AssignmentStatementParser

+ parse() : ICodeNode

ExpressionParser

+ parse() : ICodeNode
- parseExpression() : ICodeNode
- parseSimpleExpression() : ICodeNode
- parseTerm() : ICodeNode
- parseFactor() : ICodeNode

Listing 5-12: The test source file `assigments.txt`

```
BEGIN
    BEGIN {Temperature conversions.}
        five  := -1 + 2 - 3 + 4 + 3;
        ratio := five/9.0;

      fahrenheit := 72;
      centigrade := (fahrenheit - 32)*ratio;

      centigrade := 25;
      fahrenheit := centigrade/ratio + 32;

      centigrade := 25;
```

```
        fahrenheit := 32 + centigrade/ratio
    END;

    {Runtime division by zero error.}
    dze := fahrenheit/(ratio - ratio);

    BEGIN {Calculate a square root using Newton&apos;s method.}
        number := 2;
        root := number;
        root := (number/root + root)/2;
    END;

    ch  := &apos;x&apos;;
    str := &apos;hello, world&apos;;
END.
```

First modify class `PascalParserTD` in package `frontend.pascal` to serve as a superclass for the statement parsers and to parse a Pascal compound statement . [Listing 5-13](#) shows a new constructor and the new version of its `parse()` method.

[Listing 5-13:](#) A new constructor and the modified method `parse()` of class `PascalParserTD`

```
/**
 * Constructor for subclasses.
 * @param parent the parent parser.
 */
public PascalParserTD(PascalParserTD parent)
{
    super(parent.getScanner());
}

/**
 * Parse a Pascal source program and generate the symbol table
 * and the intermediate code.
 * @throws Exception if an error occurred.
 */
public void parse()
    throws Exception
{
    long startTime = System.currentTimeMillis();
    iCode = ICodeFactory.createICode();

    try {
        Token token = nextToken();
        ICodeNode rootNode = null;

        // Look for the BEGIN token to parse a compound statement.
        if (token.getType() == BEGIN) {
            StatementParser statementParser = new StatementParser(this);
            rootNode = statementParser.parse(token);
            token = currentToken();
        }
        else {
            errorHandler.flag(token, UNEXPECTED_TOKEN, this);
        }

        // Look for the final period.
        if (token.getType() != DOT) {
            errorHandler.flag(token, MISSING_PERIOD, this);
        }
        token = currentToken();

        // Set the parse tree root node.
        if (rootNode != null) {
            iCode.setRoot(rootNode);
        }

        // Send the parser summary message.
        float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
        sendMessage(new Message(PARSER_SUMMARY,
                          new Number[] {token.getLineNumber(),
                                        getErrorCount(),
                                        elapsedTime}));
    }
    catch (java.io.IOException ex) {
        errorHandler.abortTranslation(IO_ERROR, this);
    }
}
```

Whenever an instance of a Pascal parser subclass needs to create a child parser instance, it calls the new constructor which enables each child instance to inherit components dynamically from its parent instance. Using the constructor shown in [Listing 5-13](#), a child parser instance inherits the scanner from its parent. Recall that the front end factory class `FrontendFactory` calls the original constructor.

Method `parse()` checks if the current token is `BEGIN`. If so, it calls `statementParser.parse()` to parse a compound statement up to and including the matching `END` token and return the root node of the generated subtree. Afterwards, the current token should be the final period.

You'll continue to modify class `PascalParserTD` in later chapters.

Design Note

There is symmetry between extracting tokens and parsing constructs. The scanner used the first character of a token to determine what type of token to extract next from the source. After the token is extracted, the current character is the first character after the last character of the token. Similarly, the parser uses the first token of a Pascal construct (such as the `BEGIN` token of a compound statement) to determine what construct to parse next. After the construct is parsed, the current token is the first token after the last token of the construct.

Parsing Statements

[Listing 5-14](#) shows the constructor, `parse()`, and `setLineNumber()` methods of the Pascal parser subclass `StatementParser` in package `frontend.pascal.parsers`. The constructor retrieves the parsing context, as explained above.

[Listing 5-14:](#) The constructor, `parse()`, and `setLineNumber()` methods of class `StatementParser`

```
    /**
     * Constructor.
     * @param parent the parent parser.
     */
    public StatementParser(PascalParserTD parent)
    {
        super(parent);
    }

    /**
     * Parse a statement.
     * To be overridden by the specialized statement parser subclasses.
     * @param token the initial token.
     * @return the root node of the generated parse tree.
     * @throws Exception if an error occurred.
     */
    public ICodeNode parse(Token token)
        throws Exception
    {
        ICodeNode statementNode = null;

        switch ((PascalTokenType) token.getType()) {

            case BEGIN: {
                CompoundStatementParser compoundParser =
                    new CompoundStatementParser(this);
                statementNode = compoundParser.parse(token);
                break;
            }

            // An assignment statement begins with a variable&apos;s identifier.
            case IDENTIFIER: {
                AssignmentStatementParser assignmentParser =
                    new AssignmentStatementParser(this);
                statementNode = assignmentParser.parse(token);
                break;
            }

            default: {
                statementNode = ICodeFactory.createICodeNode(NO_OP);
                break;
            }
        }

        // Set the current line number as an attribute.
        setLineNumber(statementNode, token);

        return statementNode;
    }

    /**
     * Set the current line number as a statement node attribute.
     * @param node ICodeNode
     * @param token Token
     */
    protected void setLineNumber(ICodeNode node, Token token)
    {
        if (node != null) {
            node.setAttribute(LINE, token.getLineNumber());
        }
    }
}
```

Design Note

The constructor and `parse()` method of class `StatementParser` are models for the corresponding constructor and `parse()` methods of all of the statement parser subclasses.

The `parse()` method examines the token that was passed in, which must be the first token of the next statement. It uses this token to determine what statement to parse. In this chapter, the only Pascal statements the method parses are compound statements and assignment statements. Therefore, if the token is `BEGIN`, it calls `compoundParser.parse()` and if the token is an identifier, it calls `assignmentParser.parse()`. If the token is anything else, the method uses the intermediate code factory to create a `NO_OP` (no operation) node.

Methods `compoundParser.parse()` and `assignmentParser.parse()` each parses its corresponding Pascal statement and generates a parse subtree, and then returns the root node of the subtree.

Method `parse()` calls `setLineNumber()` to set the source line number attribute into the root node of the generated statement subtree, and then it returns the root node. You'll expand `parse()` to handle the other Pascal statements in Chapter 7.

Listing 5-15 shows protected method `parseList()` which parses a statement list. Each time through the `while` loop, the `parentNode` argument "adopts" as its next child the statement subtree generated by a recursive call to `parse()`. After parsing each statement, the loop looks for the semicolon token between statements and flags a `MISSING_SEMICOLON` error if necessary. The looping stops at the `terminator` token (such as `END`), which the method consumes. The method flags a syntax error (using `errorCode`) if the terminator token is missing.

Listing 5-15: Method `parseList()` of class `StatementParser`

```
    /**
     * Parse a statement list.
     * @param token the curent token.
     * @param parentNode the parent node of the statement list.
     * @param terminator the token type of the node that terminates the list.
     * @param errorCode the error code if the terminator token is missing.
     * @throws Exception if an error occurred.
     */
    protected void parseList(Token token, ICodeNode parentNode,
                             PascalTokenType terminator,
                             PascalErrorCode errorCode)
        throws Exception
    {
        // Loop to parse each statement until the END token
        // or the end of the source file.
        while (!(token instanceof EofToken) &&
               (token.getType() != terminator)) {

            // Parse a statement.  The parent node adopts the statement node.
            ICodeNode statementNode = parse(token);
```

```
        parentNode.addChild(statementNode);

        token = currentToken();
        TokenType tokenType = token.getType();

        // Look for the semicolon between statements.
        if (tokenType == SEMICOLON) {
            token = nextToken();  // consume the ;
        }

        // If at the start of the next assignment statement,
        // then missing a semicolon.
        else if (tokenType == IDENTIFIER) {
            errorHandler.flag(token, MISSING_SEMICOLON, this);
        }

        // Unexpected token.
        else if (tokenType != terminator) {
            errorHandler.flag(token, UNEXPECTED_TOKEN, this);
            token = nextToken();  // consume the unexpected token
        }
    }

    // Look for the terminator token.
    if (token.getType() == terminator) {
        token = nextToken();  // consume the terminator token
    }
    else {
        errorHandler.flag(token, errorCode, this);
    }
}
```

Parsing the Compound Statement

shows the `parse()` method of the statement parser subclass `CompoundStatementParser`.

Method `parse()` of class `CompoundStatementParser`

```
/**
 * Parse a compound statement.
 * @param token the initial token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token)
    throws Exception
{
    token = nextToken();  // consume the BEGIN

    // Create the COMPOUND node.
    ICodeNode compoundNode = ICodeFactory.createICodeNode(COMPOUND);

    // Parse the statement list terminated by the END token.
    StatementParser statementParser = new StatementParser(this);
    statementParser.parseList(token, compoundNode, END, MISSING_END);

    return compoundNode;
}
```

The method creates a COMPOUND node, which it will later return as the root of its generated subtree. As shown earlier in Figure 5-4, a COMPOUND node can have any number of ordered child nodes, and each child is the subtree generated for a statement nested within the compound statement. The method calls `statementParser.parseList()` to parse the statement list, passing END as the list-ending token type and the MISSING_END error code.

Parsing the Assignment Statement

shows the `parse()` method of the statement parser subclass `AssignmentStatementParser`. As shown in Figure 5-4, an ASSIGN node has two children. The first child is a VARIABLE node that represents the target variable on the left hand side of the `:=` token. The second child is the subtree that represents the expression on the right hand side.

Method `parse()` of class `AssignmentStatementParser`

```
/**
 * Parse an assignment statement.
 * @param token the initial token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token)
    throws Exception
{
    // Create the ASSIGN node.
    ICodeNode assignNode = ICodeFactory.createICodeNode(ASSIGN);

    // Look up the target identifer in the symbol table stack.
    // Enter the identifier into the table if it&apos;s not found.
    String targetName = token.getText().toLowerCase();
    SymTabEntry targetId = symTabStack.lookup(targetName);
    if (targetId == null) {
        targetId = symTabStack.enterLocal(targetName);
    }
    targetId.appendLineNumber(token.getLineNumber());

    token = nextToken();  // consume the identifier token

    // Create the variable node and set its name attribute.
    ICodeNode variableNode = ICodeFactory.createICodeNode(VARIABLE);
    variableNode.setAttribute(ID, targetId);

    // The ASSIGN node adopts the variable node as its first child.
    assignNode.addChild(variableNode);

    // Look for the := token.
    if (token.getType() == COLON_EQUALS) {
        token = nextToken();  // consume the :=
    }
    else {
        errorHandler.flag(token, MISSING_COLON_EQUALS, this);
```

```
        }

        // Parse the expression.  The ASSIGN node adopts the expression&apos;s
        // node as its second child.
        ExpressionParser expressionParser = new ExpressionParser(this);
        assignNode.addChild(expressionParser.parse(token));

        return assignNode;
    }
```

Because you won't be parsing variable declarations until Chapter 9, this chapter makes several major simplifications. Every variable is an identifier without array subscripts or record fields. The first time a variable appears as on the left hand side of an assignment statement, consider it "declared" and enter it into the symbol table.

Method `parse()` creates an ASSIGN node. It searches the local symbol table for the identifier whose token was passed in and enters the identifier into the symbol table if necessary. It creates a VARIABLE node and sets the ID attribute to the identifier's symbol table entry. The ASSIGN node adopts the VARIABLE node as its first child.

The method then looks for and consumes the := token, or flags a MISSING_COLON_EQUALS error. Method `expressionParser.parse()` parses the right hand side expression and returns the root of the generated expression subtree, which the ASSIGN node adopts as its second child. Finally, the ASSIGN node is the return value as the root of the assignment parse tree.

Figure 5-8 shows the generated parse tree for the following compound statement:

```
BEGIN
    alpha := 10;
    beta := 20
END
```

Figure 5-8: Parse tree for a compound statement containing assignment statements

If the source line numbers are 11 through 14, the XML representation of the parse tree is

```
<COMPOUND line="11">
    <ASSIGN line="12">
        <VARIABLE id="alpha" level="0" />
        <INTEGER_CONSTANT value="10" />
    </ASSIGN>
    <ASSIGN line="13">
        <VARIABLE id="beta" level="0" />
        <INTEGER_CONSTANT value="20" />
    </ASSIGN>
</COMPOUND>
```

The expression parser generated the INTEGER_CONSTANT nodes.

Parsing Expressions

The statement parser subclass ExpressionParser is large, but as the UML diagram in Figure 5-7 indicated, its methods closely mirror the syntax diagrams in Figure 5-2.

Listing 5-18 shows method `parse()`, which simply calls method `parseExpression()` and returns the root node of the generated expression subtree.

Listing 5-18: Method `parse()` of class `ExpressionParser`

```
/**
 * Parse an expression.
 * @param token the initial token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token)
    throws Exception
{
    return parseExpression(token);
}
```

Method `parseExpression()`

Listing 5-19 shows method `parseExpression()`, which parses an expression. According to its syntax diagram, an expression consists of a simple expression that is optionally followed by a relational Operator and a second simple expression. The method uses the static enumeration set REL_OPS which contains the Pascal relational operator tokens and the hash map REL_OPS_MAP to map each operator token to a parse tree node type. It returns the root node of the generated expression subtree.

```
    // Set of relational operators.
    private static final EnumSet<PascalTokenType> REL_OPS =
        EnumSet.of(EQUALS, NOT_EQUALS, LESS_THAN, LESS_EQUALS,
                   GREATER_THAN, GREATER_EQUALS);

    // Map relational operator tokens to node types.
    private static final HashMap<PascalTokenType, ICodeNodeType>
        REL_OPS_MAP = new HashMap<PascalTokenType, ICodeNodeType>();
    static {
        REL_OPS_MAP.put(EQUALS, EQ);
        REL_OPS_MAP.put(NOT_EQUALS, NE);
        REL_OPS_MAP.put(LESS_THAN, LT);
        REL_OPS_MAP.put(LESS_EQUALS, LE);
        REL_OPS_MAP.put(GREATER_THAN, GT);
        REL_OPS_MAP.put(GREATER_EQUALS, GE);
    };

    /**
     * Parse an expression.
     * @param token the initial token.
     * @return the root of the generated parse subtree.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseExpression(Token token)
        throws Exception
    {
        // Parse a simple expression and make the root of its tree
        // the root node.
        ICodeNode rootNode = parseSimpleExpression(token);

        token = currentToken();
        TokenType tokenType = token.getType();

        // Look for a relational operator.
        if (REL_OPS.contains(tokenType)) {

            // Create a new operator node and adopt the current tree
            // as its first child.
            ICodeNodeType nodeType = REL_OPS_MAP.get(tokenType);
            ICodeNode opNode = ICodeFactory.createICodeNode(nodeType);
            opNode.addChild(rootNode);

            token = nextToken();  // consume the operator

            // Parse the second simple expression.  The operator node adopts
            // the simple expression&apos;s tree as its second child.
            opNode.addChild(parseSimpleExpression(token));

            // The operator node becomes the new root node.
            rootNode = opNode;
        }

        return rootNode;
    }
```

Method `parseSimpleExpression()`

Method `parseSimpleExpression()` parses a simple expression, which its syntax diagram specifies consists of a term followed by zero or more terms separated by additive operators. The first term may be preceded by a + or – sign. If there was a leading – sign, the method creates a NEGATE node. The method uses the static enumeration set ADD_OPS, which contains the Pascal additive operators and the hash map ADD_OPS_OPS_MAP to map each operator token to a parse tree node type. It returns the root node of the generated simple expression subtree. See .

```
    // Set of additive operators.
    private static final EnumSet<PascalTokenType> ADD_OPS =
        EnumSet.of(PLUS, MINUS, PascalTokenType.OR);

    // Map additive operator tokens to node types.
    private static final HashMap<PascalTokenType, ICodeNodeTypeImpl>
        ADD_OPS_OPS_MAP = new HashMap<PascalTokenType, ICodeNodeTypeImpl>();
    static {
        ADD_OPS_OPS_MAP.put(PLUS, ADD);
        ADD_OPS_OPS_MAP.put(MINUS, SUBTRACT);
        ADD_OPS_OPS_MAP.put(PascalTokenType.OR, ICodeNodeTypeImpl.OR);
    };

    /**
     * Parse a simple expression.
     * @param token the initial token.
     * @return the root of the generated parse subtree.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseSimpleExpression(Token token)
        throws Exception
    {
        TokenType signType = null;  // type of leading sign (if any)

        // Look for a leading + or - sign.
        TokenType tokenType = token.getType();
        if ((tokenType == PLUS) || (tokenType == MINUS)) {
            signType = tokenType;
            token = nextToken();  // consume the + or -
        }

        // Parse a term and make the root of its tree the root node.
        ICodeNode rootNode = parseTerm(token);

        // Was there a leading - sign?
        if (signType == MINUS) {

            // Create a NEGATE node and adopt the current tree
            // as its child. The NEGATE node becomes the new root node.
            ICodeNode negateNode = ICodeFactory.createICodeNode(NEGATE);
            negateNode.addChild(rootNode);
            rootNode = negateNode;
        }

        token = currentToken();
        tokenType = token.getType();
```

```
                // Loop over additive operators.
                while (ADD_OPS.contains(tokenType)) {

                    // Create a new operator node and adopt the current tree
                    // as its first child.
                    ICodeNodeType nodeType = ADD_OPS_OPS_MAP.get(tokenType);
                    ICodeNode opNode = ICodeFactory.createICodeNode(nodeType);
                    opNode.addChild(rootNode);

                    token = nextToken();  // consume the operator

                    // Parse another term.  The operator node adopts
                    // the term&apos;s tree as its second child.
                    opNode.addChild(parseTerm(token));

                    // The operator node becomes the new root node.
                    rootNode = opNode;

                    token = currentToken();
                    tokenType = token.getType();
                }

                return rootNode;
        }
```

Each time the `while` loop finds an additive operator, it uses `ADD_OPS_OPS_MAP` to create an operator node of the appropriate type. The loop exits when there are no more additive operators. Each subsequent operator node adopts the current value of `rootNode` as its first child. In other words, the expression subtree built so far is "pushed down" to the left to become the new operator's first operand. Figure 5-9 shows the progression of building the parse tree for the simple expression `a+b+c+d+e`.

When the final parse tree is executed, the operands of the simple expression are evaluated in order.[4] This follows the precedence rule described earlier that states that operators at the same level execute from left to right. The XML representation of the final expression tree is
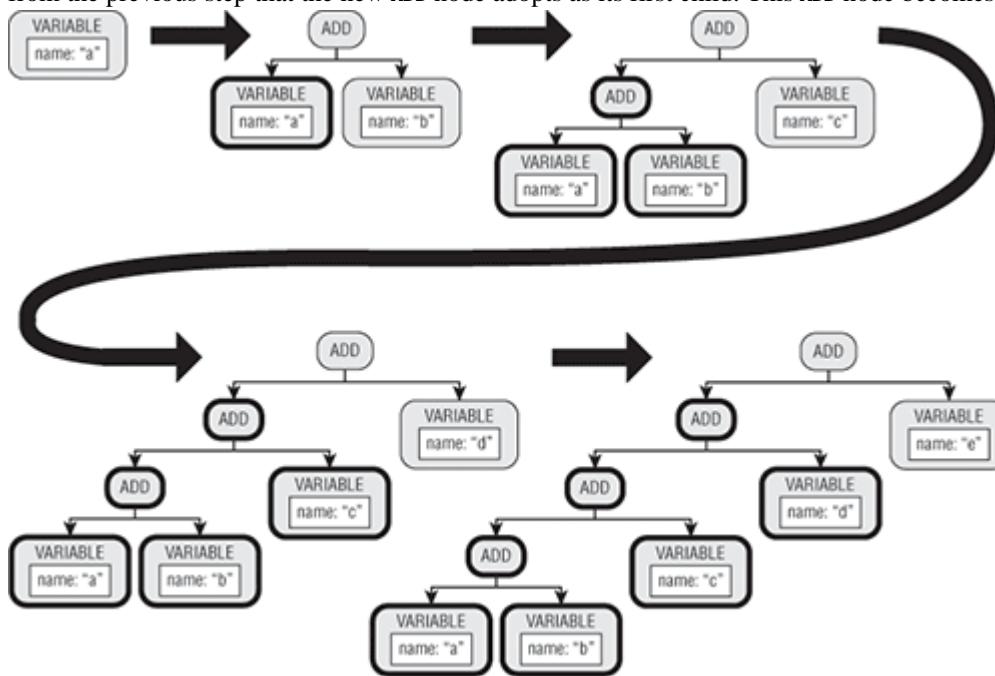
[4] You'll see in the next chapter that this is accomplished by a postorder traversal of the parse tree.

```
        <ADD>
            <ADD>
                <ADD>
                    <ADD>
                        <VARIABLE id="a" level="0" />
                        <VARIABLE id="b" level="0" />
                    </ADD>
                    <VARIABLE id="c" level="0" />
                </ADD>
                <VARIABLE id="d" level="0" />
            </ADD>
            <VARIABLE id="e" level="0" />
        </ADD>
```
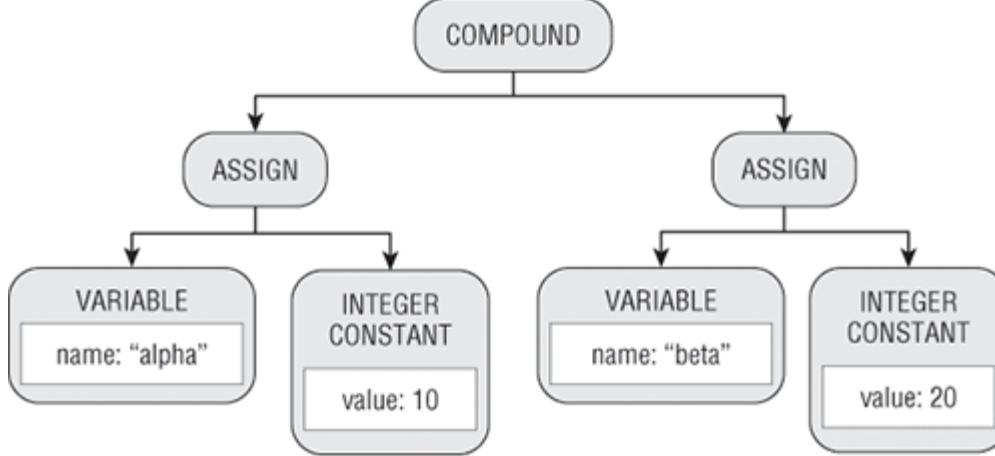
Figure 5-9: How to build the parse subtree for the expression `a+b+c+d+e`. In each step after the first, the nodes in bold are the subtree from the previous step that the new ADD node adopts as its first child. This ADD node becomes the new root node.

Method `parseTerm( )`

Method `parseTerm()` is similar to method `parseSimpleExpression()`. It parses a term, which its syntax diagram specifies is a factor followed by zero or more factors separated by multiplicative operators. The method uses the static enumeration set `MULT_OPS`, which contains the Pascal multiplicative operators and the hash map `MULT_OPS_MAP` to map each operator token to a parse tree node type. It returns the root node of the generated term subtree. See Listing 5-21.

Listing 5-21: Method `parseTerm()` of class `ExpressionParser`

```
    // Set of multiplicative operators.
    private static final EnumSet<PascalTokenType> MULT_OPS =
        EnumSet.of(STAR, SLASH, DIV, PascalTokenType.MOD, PascalTokenType.AND);

    // Map multiplicative operator tokens to node types.
    private static final HashMap<PascalTokenType, ICodeNodeType>
```

```
    MULT_OPS_OPS_MAP = new HashMap<PascalTokenType, ICodeNodeType>();
static {
    MULT_OPS_OPS_MAP.put(STAR, MULTIPLY);
    MULT_OPS_OPS_MAP.put(SLASH, FLOAT_DIVIDE);
    MULT_OPS_OPS_MAP.put(DIV, INTEGER_DIVIDE);
    MULT_OPS_OPS_MAP.put(PascalTokenType.MOD, ICodeNodeTypeImpl.MOD);
    MULT_OPS_OPS_MAP.put(PascalTokenType.AND, ICodeNodeTypeImpl.AND);
};

/**
 * Parse a term.
 * @param token the initial token.
 * @return the root of the generated parse subtree.
 * @throws Exception if an error occurred.
 */
private ICodeNode parseTerm(Token token)
    throws Exception
{
    // Parse a factor and make its node the root node.
    ICodeNode rootNode = parseFactor(token);

    token = currentToken();
    TokenType tokenType = token.getType();

    // Loop over multiplicative operators.
    while (MULT_OPS.contains(tokenType)) {

        // Create a new operator node and adopt the current tree
        // as its first child.
        ICodeNodeType nodeType = MULT_OPS_OPS_MAP.get(tokenType);
        ICodeNode opNode = ICodeFactory.createICodeNode(nodeType);
        opNode.addChild(rootNode);

        token = nextToken();  // consume the operator

        // Parse another factor.  The operator node adopts
        // the term&apos;s tree as its second child.
        opNode.addChild(parseFactor(token));

        // The operator node becomes the new root node.
        rootNode = opNode;

        token = currentToken();
        tokenType = token.getType();
    }

    return rootNode;
}
```

Method `parseFactor()`

Method `parseFactor()`, shown in [Listing 5-22](#), parses a factor. According to its syntax diagram, a factor can be a variable, a number, a string, NOT followed by another factor, or a parenthesized expression (as explained in the list following [Listing 5-22](#)).

[Listing 5-22:](#) **Method `parseFactor()` of class `ExpressionParser`**

```
/**
 * Parse a factor.
 * @param token the initial token.
 * @return the root of the generated parse subtree.
 * @throws Exception if an error occurred.
 */
private ICodeNode parseFactor(Token token)
    throws Exception
{
    TokenType tokenType = token.getType();
    ICodeNode rootNode = null;

    switch ((PascalTokenType) tokenType) {

        case IDENTIFIER: {
            // Look up the identifier in the symbol table stack.
            // Flag the identifier as undefined if it&apos;s not found.
            String name = token.getText().toLowerCase();
            SymTabEntry id = symTabStack.lookup(name);
            if (id == null) {
                errorHandler.flag(token, IDENTIFIER_UNDEFINED, this);
                id = symTabStack.enterLocal(name);
            }

            rootNode = ICodeFactory.createICodeNode(VARIABLE);
            rootNode.setAttribute(ID, id);
            id.appendLineNumber(token.getLineNumber());

            token = nextToken();  // consume the identifier
            break;
        }

        case INTEGER: {
            // Create an INTEGER_CONSTANT node as the root node.
            rootNode = ICodeFactory.createICodeNode(INTEGER_CONSTANT);
            rootNode.setAttribute(VALUE, token.getValue());

            token = nextToken();  // consume the number
            break;
        }

        case REAL: {
            // Create an REAL_CONSTANT node as the root node.
            rootNode = ICodeFactory.createICodeNode(REAL_CONSTANT);
            rootNode.setAttribute(VALUE, token.getValue());

            token = nextToken();  // consume the number
            break;
        }

        case STRING: {
            String value = (String) token.getValue();

            // Create a STRING_CONSTANT node as the root node.
            rootNode = ICodeFactory.createICodeNode(STRING_CONSTANT);
            rootNode.setAttribute(VALUE, value);

            token = nextToken();  // consume the string
            break;
        }
```

```
        case NOT: {
            token = nextToken();  // consume the NOT

            // Create a NOT node as the root node.
            rootNode = ICodeFactory.createICodeNode(ICodeNodeTypeImpl.NOT);

            // Parse the factor.  The NOT node adopts the
            // factor node as its child.
            rootNode.addChild(parseFactor(token));

            break;
        }

        case LEFT_PAREN: {
            token = nextToken();      // consume the (

            // Parse an expression and make its node the root node.
            rootNode = parseExpression(token);

            // Look for the matching ) token.
            token = currentToken();
            if (token.getType() == RIGHT_PAREN) {
                token = nextToken();  // consume the )
            }
            else {
                errorHandler.flag(token, MISSING_RIGHT_PAREN, this);
            }

            break;
        }

        default: {
            errorHandler.flag(token, UNEXPECTED_TOKEN, this);
            break;
        }
    }

    return rootNode;
}
```

If the current token is an identifier, method `parseFactor()` first looks up the variable name in the local symbol table. If it's not in there (which in this chapter means that the variable did not already appear on the left hand side of an assignment statement), the method flags an `IDENTIFIER_UNDEFINED` error and enters it into the symbol table. Otherwise, the method creates a `VARIABLE` node and sets the `ID` attribute to the variable's symbol table entry. Future chapters will expand upon parsing an expression identifier.

If the current token is a number, the method creates an `INTEGER_CONSTANT` node or a `REAL_CONSTANT` node, depending on the number's data type. The method sets the `VALUE` attribute to the number's value which is either an `Integer` or a `Float` object. If the current token is a string, the method creates a `STRING_CONSTANT` node and sets the `VALUE` attribute to the string value.

If the current token is `NOT`, method `parseFactor()` creates a `NOT` node and calls itself recursively. The `NOT` node adopts the factor parse tree as its only child.

If the current token is `(`, method `parseFactor()` consumes the `(` and recursively calls method `parseExpression()` to parse the enclosed expression. It then looks for and consumes the matching `)` token or flags a `MISSING_RIGHT_PAREN` error.

If the current token is any other token, method `parseFactor()` flags an `UNEXPECTED_TOKEN` error and returns null. Otherwise, it returns the root node of the generated factor subtree.

Design Note

It should now be clear why this style of parsing is called top-down. The parser starts at the topmost construct (which for now is the compound statement) and works its way down to the bottommost construct. Our top-down parser is also a recursive descent parser. Because statements and expressions can be nested inside each other, the parsing methods call each other recursively as they work their way down.

Program 5: Pascal Syntax Checker I

Add the statements

```
if (intermediate) {
    ParseTreePrinter treePrinter =
                     new ParseTreePrinter(System.out);
    treePrinter.print(iCode);
}
```

to the constructor of the main `Pascal` class to print the parse tree using the utility class `ParseTreePrinter` which you saw earlier in .

Since you haven't yet made any changes to the back end, the back end compiler and interpreter still do nothing more than write summary messages. So basically, what you have is a syntax checker utility that parses the source program, builds the symbol table, and generates the parse tree. In the next chapter, you will develop executors in the interpreter back end to execute the generated parse trees.

Assuming the class files are in the `classes` directory and the source file `assignments.txt` (see) is in the current directory, a command line similar to

```
java -classpath classes Pascal execute -i assignments.txt
```

will parse the "program" in `assignments.txt`. The `-i` argument specifies printing the intermediate code, which is the XML rendition of the entire generated parse tree if there were no syntax errors. shows the output.

Listing 5-23: Output listing when there are no syntax errors

```
001 BEGIN
002     BEGIN {Temperature conversions.}
003         five  := -1 + 2 - 3 + 4 + 3;
004         ratio := five/9.0;
005
```

```
006          fahrenheit := 72;
007          centigrade := (fahrenheit - 32)*ratio;
008
009          centigrade := 25;
010          fahrenheit := centigrade/ratio + 32;
011
012          centigrade := 25;
013          fahrenheit := 32 + centigrade/ratio
014      END;
015
016      {Runtime division by zero error.}
017      dze := fahrenheit/(ratio - ratio);
018
019      BEGIN {Calculate a square root using Newton&apos;s method.}
020          number := 2;
021          root := number;
022          root := (number/root + root)/2;
023      END;
024
025      ch  := &apos;x&apos;;
026      str := &apos;hello, world&apos;
027 END.

                  27 source lines.
                   0 syntax errors.
                0.05 seconds total parsing time.


===== INTERMEDIATE CODE =====

<COMPOUND line="1">
    <COMPOUND line="2">
        <ASSIGN line="3">
            <VARIABLE id="five" level="0" />
            <ADD>
                <ADD>
                    <SUBTRACT>
                        <ADD>
                            <NEGATE>
                                <INTEGER_CONSTANT value="1" />
                            </NEGATE>
                            <INTEGER_CONSTANT value="2" />
                        </ADD>
                        <INTEGER_CONSTANT value="3" />
                    </SUBTRACT>
                    <INTEGER_CONSTANT value="4" />
                </ADD>
                <INTEGER_CONSTANT value="3" />
            </ADD>
        </ASSIGN>
        <ASSIGN line="4">
            <VARIABLE id="ratio" level="0" />
            <FLOAT_DIVIDE>
                <VARIABLE id="five" level="0" />
                <REAL_CONSTANT value="9.0" />
            </FLOAT_DIVIDE>
        </ASSIGN>
        <ASSIGN line="6">
            <VARIABLE id="fahrenheit" level="0" />
            <INTEGER_CONSTANT value="72" />
        </ASSIGN>
        <ASSIGN line="7">
            <VARIABLE id="centigrade" level="0" />
            <MULTIPLY>
                <SUBTRACT>
                    <VARIABLE id="fahrenheit" level="0" />
                    <INTEGER_CONSTANT value="32" />
                </SUBTRACT>
                <VARIABLE id="ratio" level="0" />
            </MULTIPLY>
        </ASSIGN>
        <ASSIGN line="9">
            <VARIABLE id="centigrade" level="0" />
            <INTEGER_CONSTANT value="25" />
        </ASSIGN>
        <ASSIGN line="10">
            <VARIABLE id="fahrenheit" level="0" />
            <ADD>
                <FLOAT_DIVIDE>
                    <VARIABLE id="centigrade" level="0" />
                    <VARIABLE id="ratio" level="0" />
                </FLOAT_DIVIDE>
                <INTEGER_CONSTANT value="32" />
            </ADD>
        </ASSIGN>
        <ASSIGN line="12">
            <VARIABLE id="centigrade" level="0" />
            <INTEGER_CONSTANT value="25" />
        </ASSIGN>
        <ASSIGN line="13">
            <VARIABLE id="fahrenheit" level="0" />
            <ADD>
                <INTEGER_CONSTANT value="32" />
                <FLOAT_DIVIDE>
                    <VARIABLE id="centigrade" level="0" />
                    <VARIABLE id="ratio" level="0" />
                </FLOAT_DIVIDE>
            </ADD>
        </ASSIGN>
    </COMPOUND>
    <ASSIGN line="17">
        <VARIABLE id="dze" level="0" />
        <FLOAT_DIVIDE>
            <VARIABLE id="fahrenheit" level="0" />
            <SUBTRACT>
                <VARIABLE id="ratio" level="0" />
                <VARIABLE id="ratio" level="0" />
            </SUBTRACT>
        </FLOAT_DIVIDE>
    </ASSIGN>
    <COMPOUND line="19">
        <ASSIGN line="20">
```

```
                <VARIABLE id="number" level="0" />
                <INTEGER_CONSTANT value="2" />
            </ASSIGN>
            <ASSIGN line="21">
                <VARIABLE id="root" level="0" />
                <VARIABLE id="number" level="0" />
            </ASSIGN>
            <ASSIGN line="22">
                <VARIABLE id="root" level="0" />
                <FLOAT_DIVIDE>
                    <ADD>
                        <FLOAT_DIVIDE>
                            <VARIABLE id="number" level="0" />
                            <VARIABLE id="root" level="0" />
                        </FLOAT_DIVIDE>
                        <VARIABLE id="root" level="0" />
                    </ADD>
                    <INTEGER_CONSTANT value="2" />
                </FLOAT_DIVIDE>
            </ASSIGN>
        </COMPOUND>
        <ASSIGN line="25">
            <VARIABLE id="ch" level="0" />
            <STRING_CONSTANT value="x" />
        </ASSIGN>
        <ASSIGN line="26">
            <VARIABLE id="str" level="0" />
            <STRING_CONSTANT value="hello, world" />
        </ASSIGN>
    </COMPOUND>
</COMPOUND>

                0 statements executed.
                0 runtime errors.
             0.00 seconds total execution time.
```

Listing 5-24 shows the output when there are syntax errors.

Listing 5-24: Output listing with syntax errors

```
001 BEGIN
002     BEGIN {Temperature conversions.}
003         five  := -1 + 2 - 3 + 4 - -3;
                              ^
*** Unexpected token [at "-"]
004         ratio := five/9.0;
005
006         fahrenheit := 72;
007         centigrade := (((fahrenheit - 32)))*ratio;
008
009         centigrade := 25;;;
010         fahrenheit := centigrade/ratio + 32;
011
012         centigrade := 25
013         fahrenheit := 32 + centigrade/ratio;
                ^
*** Missing ; at "fahrenheit"]
014
015         centigrade := 25;
016         fahrenheit := celsius/ratio + 32
                          ^
*** Undefined identifier [at "celsius"]
017     END
018
019     dze fahrenheit/((ratio - ratio) := ;
            ^
*** Missing ; [at "dze"]
                ^
*** Missing := [at "fahrenheit"]
                                  ^
*** Missing ) [at ":="]
                                  ^
*** Unexpected token [at ":="]
020
021 END.

                21 source lines.
                 7 syntax errors.
             0.06 seconds total parsing time.
```

Chapter 6

Interpreting Expressions and Assignment Statements

In the previous chapter, you developed parsers that generated intermediate code in the form of parse trees for Pascal compound and assignment statements and for expressions. In this chapter, you'll develop code in the back end to interpret a parse tree in order to execute these statements and expressions.

Goals and Approach

This chapter picks up where the previous chapter left off. It has one major goal:

- Language-independent executors in the interpreter back end that will interpret the intermediate code and execute expressions and compound and assignment statements.

In the previous chapter, you developed the Pascal parser subclass `StatementParser` in the front end and its subclasses `CompoundStatementParser`, `AssignmentStatementParser`, and `ExpressionParser` to parse the Pascal statements and expressions. In a similar fashion, the approach for this chapter will be to develop executor subclasses in the interpreter back end. However, these executor subclasses will be language-independent. At the end of the chapter, to verify your work, a simple interpreter utility program will read and execute Pascal assignment statements and expressions.

Runtime Error Handling

The Pascal-specific parsing routines in package `frontend.pascal` use the error handler class `PascalErrorHandler` and the enumerated type `PascalErrorCode`. Similarly, the language-independent interpreter in package `backend.interpreter` uses a runtime error handler class `RuntimeErrorHandler` and the enumerated type `RuntimeErrorCode`.[1]

[1] By *run time* (noun) and *runtime* (adjective), we mean the time when the back end is executing the source program.

Listing 6-1 shows the `flag()` method of class `RuntimeErrorHandler` in package `backend.interpreter`.

Listing 6-1: Method `flag()` of class `RuntimeErrorHandler`

```
/**
 * <h1>RuntimeErrorHandler</h1>
 *
 * <p>Runtime error handler for the backend interpreter.</p>
 */
public class RuntimeErrorHandler
{
    private static final int MAX_ERRORS = 5;

    private static int errorCount = 0;    // count of runtime errors

    /**
     * Flag a runtime error.
     * @param node the root node of the offending statement or expression.
     * @param errorCode the runtime error code.
     * @param backend the backend processor.
     */
    public void flag(ICodeNode node, RuntimeErrorCode errorCode,
                     Backend backend)
    {
        String lineNumber = null;

        // Look for the ancestor statement node with a line number attribute.
        while ((node != null) && (node.getAttribute(LINE) == null)) {
            node = node.getParent();
        }

        // Notify the interpreter&apos;s listeners.
        backend.sendMessage(
            new Message(RUNTIME_ERROR,
            new Object[] {errorCode.toString(),
                          (Integer) node.getAttribute(LINE)}));

        if (++errorCount > MAX_ERRORS) {
            System.out.println("*** ABORTED AFTER TOO MANY RUNTIME ERRORS.");
            System.exit(-1);
        }
    }
}
```

Whenever the interpreter detects a runtime error, it sends an error message to all the listeners of the back end. By convention, this message has the format

RUNTIME_ERROR Message	
`errorCode.toString()`	runtime error message
`node.getAttribute(LINE)`	source line number

Method `flag()` constructs these messages. It is passed the current parse tree node, and starting with that node, it searches up the parent chain looking for the nearest node that has a `LINE` attribute. That would be the line number of the source statement that caused the runtime error.

Listing 6-2 shows the enumerated type `RuntimeErrorCode`. The enumerated values represent the runtime errors that the interpreter is able to detect.

Listing 6-2: Enumerated type **`RuntimeErrorCode`**

```
package wci.backend.interpreter;

/**
 * <h1>RuntimeErrorCode</h1>
 *
 * <p>Runtime error codes.</p>
 */
public enum RuntimeErrorCode
{
    UNINITIALIZED_VALUE("Uninitialized value"),
    VALUE_RANGE("Value out of range"),
    INVALID_CASE_EXPRESSION_VALUE("Invalid CASE expression value"),
    DIVISION_BY_ZERO("Division by zero"),
    INVALID_STANDARD_FUNCTION_ARGUMENT("Invalid standard function argument"),
    INVALID_INPUT("Invalid input"),
    STACK_OVERFLOW("Runtime stack overflow"),
    UNIMPLEMENTED_FEATURE("Unimplemented runtime feature");

    private String message;  // error message

    /**
     * Constructor.
     * @param message the error message.
     */
    RuntimeErrorCode(String message)
    {
        this.message = message;
    }

    public String toString()
    {
        return message;
    }
}
```

Executing Assignment Statements and Expressions

In Chapter 5, you parsed Pascal expressions and compound and assignment statements and generated language-independent intermediate code in the form of a parse tree. Now you're ready to write language-independent classes in the back end to interpret the parse trees in order to execute these statements and expressions.
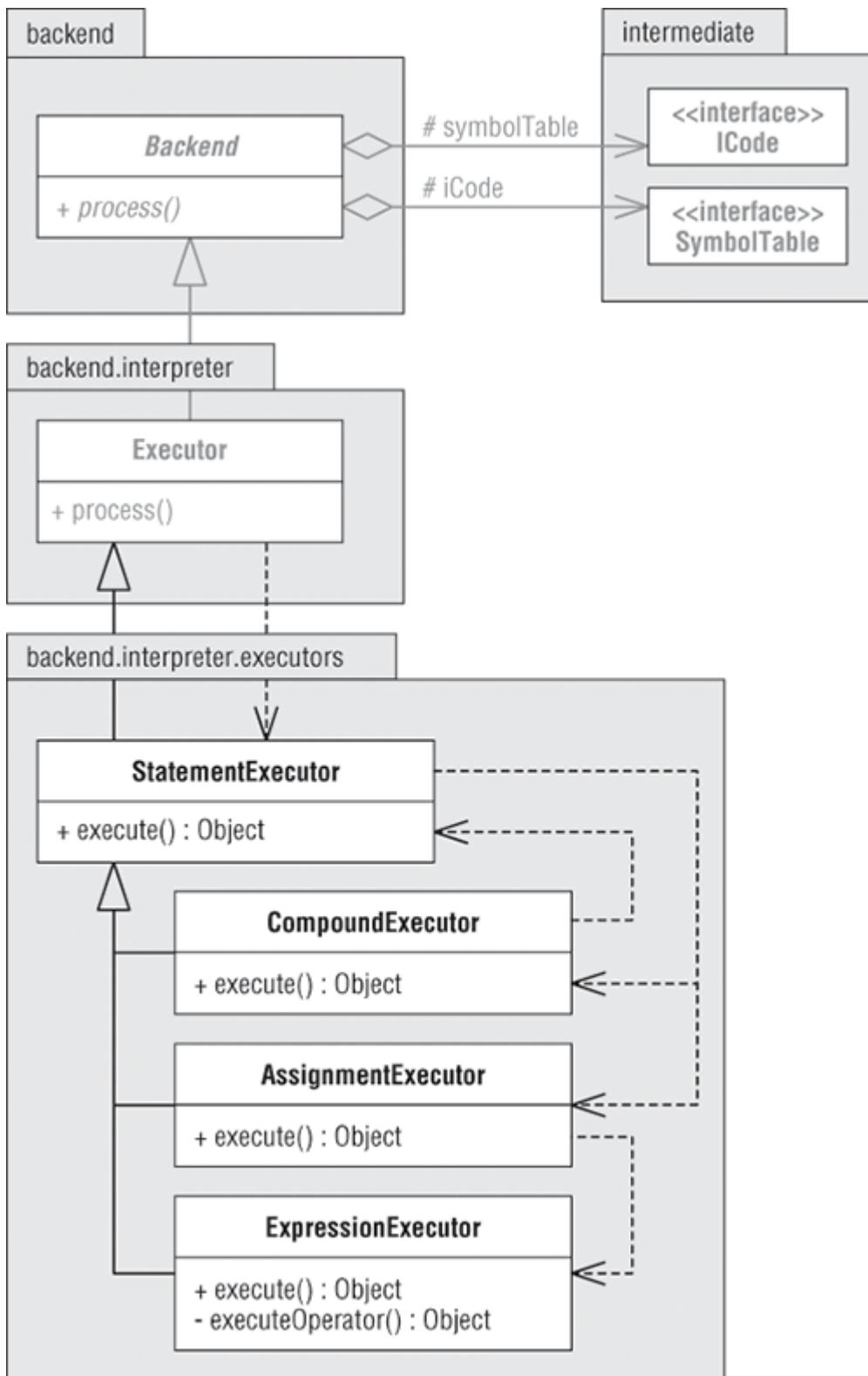
The Statement Executor Subclasses

Figure 6-1 shows the UML diagram of the executor classes in package `backend.interpreter.executors`. Class `Executor`, which you first saw in Chapter 2, is now the base class of class `StatementExecutor`, which in turn is the base class of classes `CompoundExecutor` and `ExpressionExecutor`. Compare this diagram to the one in Figure 5-7.

Design Note

Well-designed software often exhibit architectural symmetries. Figure 2-3 in Chapter 2 showed how the `Parser` and `Backend` framework classes were symmetrical about the intermediate code and symbol table components. Figures 5-7 and 6-1 show that class `PascalParserTD` and its subclasses are symmetrical with class `Executor` and its subclasses.

Figure 6-1: Executor subclass `StatementExecutor` and its subclasses

Class `StatementExecutor` and each of its subclasses has an `execute()` method specialized to execute a particular language construct by interpreting that construct's parse tree. Its return value is always null, except for class `ExpressionExecutor`, whose return value is the computed value of the expression. Since the intermediate code and the symbol table are both language-independent, `StatementExecutor` and its subclasses will also all be language-independent.[2]

---

[2] Since you will only interpret Pascal programs in this book, your `Executor` subclasses will have a Pascal bias. However, the goal remains for the back end to be able to interpret more than one source language.

Listing 6-3 shows a new constructor for class `Executor` and a new version of its `process()` method.

Listing 6-3: A constructor and method `process()` of class `Executor`

```
protected static int executionCount;
protected static RuntimeErrorHandler errorHandler;

static {
    executionCount = 0;
    errorHandler = new RuntimeErrorHandler();
}

/**
 * Constructor for subclasses.
```

```
 * @param the parent executor.
 */
public Executor(Executor parent)
{
    super();
}

/**
 * Execute the source program by processing the intermediate code
 * and the symbol table stack generated by the parser.
 * @param iCode the intermediate code.
 * @param symTabStack the symbol table stack.
 * @throws Exception if an error occurred.
 */
public void process(ICode iCode, SymTabStack symTabStack)
    throws Exception
{
    this.symTabStack = symTabStack;
    this.iCode = iCode;

    long startTime = System.currentTimeMillis();

    // Get the root node of the intermediate code and execute.
    ICodeNode rootNode = iCode.getRoot();
    StatementExecutor statementExecutor = new StatementExecutor(this);
    statementExecutor.execute(rootNode);

    float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
    int runtimeErrors = errorHandler.getErrorCount();

    // Send the interpreter summary message.
    sendMessage(new Message(INTERPRETER_SUMMARY,
                            new Number[] {executionCount,
                                          runtimeErrors,
                                          elapsedTime}));
}
```

All the executor subclasses call the new constructor. Method `process()` obtains the root of the intermediate code generated by the front end parser and calls `statementExecutor.execute()`.

Executing Statements

Listing 6-4 shows the constructor and the `execute()` method of the executor subclass `StatementExecutor`. Also shown is its private `sendSourceLineMessage()` method.

Listing 6-4: The constructor and methods `execute()` and `sendMessage()` of class `StatementExecutor`

```
/**
 * Constructor.
 * @param the parent executor.
 */
public StatementExecutor(Executor parent)
{
    super(parent);
}

/**
 * Execute a statement.
 * To be overridden by the specialized statement executor subclasses.
 * @param node the root node of the statement.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    ICodeNodeTypeImpl nodeType = (ICodeNodeTypeImpl) node.getType();

    // Send a message about the current source line.
    sendSourceLineMessage (node);

    switch (nodeType) {

        case COMPOUND: {
            CompoundExecutor compoundExecutor = new CompoundExecutor(this);
            return compoundExecutor.execute(node);
        }

        case ASSIGN: {
            AssignmentExecutor assignmentExecutor =
                new AssignmentExecutor(this);
            return assignmentExecutor.execute(node);
        }

        case NO_OP: return null;

        default: {
            errorHandler.flag(node, UNIMPLEMENTED_FEATURE, this);
            return null;
        }
    }
}

/**
 * Send a message about the current source line.
 * @param node the statement node.
 */
private void sendSourceLineMessage(ICodeNode node)
{
    Object lineNumber = node.getAttribute(LINE);

    // Send the SOURCE_LINE message.
    if (lineNumber != null) {
        sendMessage(new Message(SOURCE_LINE, lineNumber));
    }
}
```

Design Note

The constructor and `execute()` method of class `StatementExecutor` are models for the corresponding constructors and `execute()` methods of all of the statement executor subclasses.

The `execute()` method first calls `sendSourceLineMessage()` to send a `SOURCE_LINE` message to its listeners. By convention, a `SOURCE_LINE` message has the format

SOURCE_LINE Message	
`node.getAttribute(LINE)`	source line number

This message indicates the source line number of the statement that is about to be executed, which can be very useful for a debugger to implement tracing. The front end will ignore this message for now.

After sending the message, method `execute()` calls `compoundExecutor.execute()` or `assignmentExecutor.execute()` or does nothing depending on whether the node that was passed in is a `COMPOUND`, `ASSIGN`, or `NO_OP` node, respectively. If the node is not one of those types, the method flags an `UNIMPLEMENTED_FEATURE` runtime error. This method will change in later chapters when you add other types of statements to execute.

Executing the Compound Statement

[Listing 6-5](#) shows the `execute()` method of the statement executor subclass `CompoundExecutor`. The method loops over the child nodes of the `COMPOUND` node and calls `statementExecutor.execute()` for each child.

Listing 6-5: Method `execute()` of class `CompoundExecutor`

```
/**
 * Execute a compound statement.
 * @param node the root node of the compound statement.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    // Loop over the children of the COMPOUND node and execute each child.
    StatementExecutor statementExecutor = new StatementExecutor(this);
    ArrayList<ICodeNode> children = node.getChildren();
    for (ICodeNode child : children) {
        statementExecutor.execute(child);
    }

    return null;
}
```

Executing the Assignment Statement

[Listing 6-6](#) shows the `execute()` and `sendMessage()` methods of the statement executor subclass `AssignmentExecutor`.

Listing 6-6: Methods `execute()` and `sendMessage()` of class `AssignmentExecutor`

```
/**
 * Execute an assignment statement.
 * @param node the root node of the statement.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    // The ASSIGN node&apos;s children are the target variable
    // and the expression.
    ArrayList<ICodeNode> children = node.getChildren();
    ICodeNode variableNode = children.get(0);
    ICodeNode expressionNode = children.get(1);

    // Execute the expression and get its value.
    ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
    Object value = expressionExecutor.execute(expressionNode);

    // Set the value as an attribute of the variable&apos;s symbol table entry.
    SymTabEntry variableId = (SymTabEntry) variableNode.getAttribute(ID);
    variableId.setAttribute(DATA_VALUE, value);

    sendMessage(node, variableId.getName(), value);

    ++executionCount;
    return null;
}

/**
 * Send a message about the assignment operation.
 * @param node the ASSIGN node.
 * @param variableName the name of the target variable.
 * @param value the value of the expression.
 */
private void sendMessage(ICodeNode node, String variableName, Object value)
{
    Object lineNumber = node.getAttribute(LINE);

    // Send an ASSIGN message.
    if (lineNumber != null) {
        sendMessage(new Message(ASSIGN, new Object[] {lineNumber,
                                                      variableName,
                                                      value}));
    }
}
```

Method `execute()` obtains the two child nodes of the `ASSIGN` node. The first child is the `VARIABLE` node of the target variable of the assignment and the second child is the root of the expression subtree. It calls method `expressionExecutor.execute()` to obtain the computed value of the expression, which it sets in the `DATA_VALUE` attribute of the target variable's symbol table entry. Method `execute()` calls `sendMessage()` to send a message to the executor's listeners about the assignment.

By convention, the assignment message has the format

ASSIGN Message	
`node.getAttribute(LINE)`	source line number
`variableName`	target variable name
`value`	expression value

This message can be useful for debugging and tracing.

Executing Expressions

Listing 6-7 shows the `execute()` method of the statement executor subclass `ExpressionExecutor`.

Listing 6-7: Method `execute()` of class `ExpressionExecutor`

```
/**
 * Execute an expression.
 * @param node the root intermediate code node of the compound statement.
 * @return the computed value of the expression.
 */
public Object execute(ICodeNode node)
{
    ICodeNodeTypeImpl nodeType = (ICodeNodeTypeImpl) node.getType();

    switch (nodeType) {

        case VARIABLE: {

            // Get the variable's symbol table entry and return its value.
            SymTabEntry entry = (SymTabEntry) node.getAttribute(ID);
            return entry.getAttribute(DATA_VALUE);
        }

        case INTEGER_CONSTANT: {

            // Return the integer value.
            return (Integer) node.getAttribute(VALUE);
        }

        case REAL_CONSTANT: {

            // Return the float value.
           return (Float) node.getAttribute(VALUE);
        }

        case STRING_CONSTANT: {

            // Return the string value.
            return (String) node.getAttribute(VALUE);
        }

        case NEGATE: {

            // Get the NEGATE node's expression node child.
            ArrayList<ICodeNode> children = node.getChildren();
            ICodeNode expressionNode = children.get(0);

            // Execute the expression and return the negative of its value.
            Object value = execute(expressionNode);
            if (value instanceof Integer) {
                return -((Integer) value);
            }
            else {
                return -((Float) value);
            }
        }

        case NOT: {

            // Get the NOT node's expression node child.
            ArrayList<ICodeNode> children = node.getChildren();
            ICodeNode expressionNode = children.get(0);

            // Execute the expression and return the "not" of its value.
            boolean value = (Boolean) execute(expressionNode);
            return !value;
        }

        // Must be a binary operator.
        default: return executeBinaryOperator(node, nodeType);
    }
}
```

Operator Precedence

The syntax diagrams in Figure 5-2 incorporated Pascal's operator precedence rules. Class `ExpressionParser` in the front end builds expression subtrees that mirror these syntax diagrams. Therefore, the precedence rules can take care of themselves during source program execution if the expression executor evaluates the tree nodes in the proper order.

An expression subtree is a binary tree, and the expression executor must do a *postorder traversal*[3] of the nodes to evaluate them. If the root node of a subtree is an operator node, the executor first executes the node's left child to get the computed value of the first operand. Then if the operator node has a right child, the executor executes the right child to get the value of the second operand. Finally, the executor examines the operator node and performs the specified operation on the one or two operand values. If either child node is the root of a subtree, the executor recursively does a postorder traversal on the subtree.
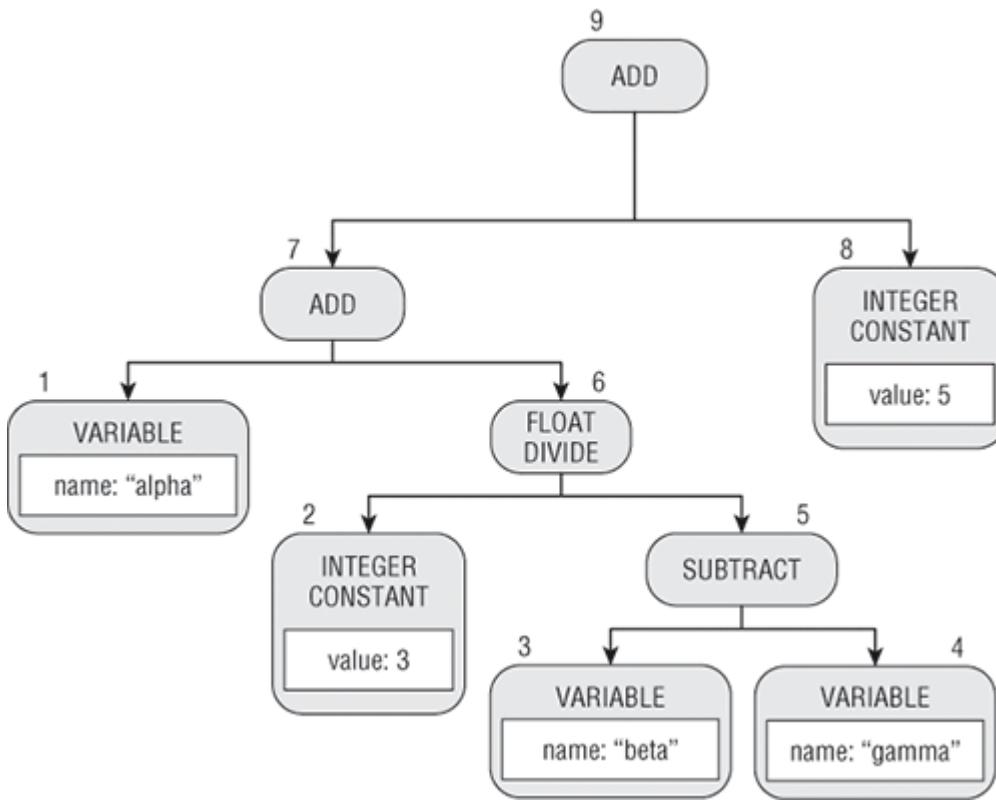
[3] In a *postorder* tree traversal, you visit a node *after* visiting its children.

Figure 6-2 shows the parse tree for the expression

```
alpha + 3/(beta - gamma) + 5
```

with numbers indicating the order that the executor will evaluate the nodes.

Figure 6-2: An expression parse tree showing the order that the nodes are evaluated

Operand Values

If the node that was passed in is a VARIABLE node, method execute() returns the variable's value that it obtains from the VALUE attribute of the variable's symbol table entry. If the node is an INTEGER_CONSTANT node or a REAL_CONSTANT node, the method retrieves the Integer or Float value from the node's VALUE attribute, respectively. If the node is a STRING_CONSTANT node, the method retrieves and returns the node's string value.

If the node is a NEGATE node, method execute() obtains the node's child expression subtree. It recursively calls itself to evaluate the expression and then it returns the negative of the value. Similarly, if the node is a NOT node, the method obtains the node's child expression subtree. It calls itself to evaluate the expression and then it returns the boolean *not* of the value.

If the node is a binary operator node, method execute() calls method executeBinaryOperator() and returns the computed value. See Listing 6-8. Method executeBinaryOperator() uses the static enumeration set ARITH_OPS which contains all the node types that represent arithmetic operators. The method first recursively calls method execute() to execute the two children of the node to obtain the values of the first and second operands. It sets local variable integerMode to true if and only if both operand values are integer.

Listing 6-8: Method `executeBinaryOperator()` of class `ExpressionExecutor`

```
    // Set of arithmetic operator node types.
    private static final EnumSet<ICodeNodeTypeImpl> ARITH_OPS =
        EnumSet.of(ADD, SUBTRACT, MULTIPLY, FLOAT_DIVIDE, INTEGER_DIVIDE);

    /**
     * Execute a binary operator.
     * @param node the root node of the expression.
     * @param nodeType the node type.
     * @return the computed value of the expression.
     */
    private Object executeBinaryOperator(ICodeNode node,
                                         ICodeNodeTypeImpl nodeType)
    {
        // Get the two operand children of the operator node.
        List<ICodeNode> children = node.getChildren();
        ICodeNode operandNode1 = children.get(0);
        ICodeNode operandNode2 = children.get(1);

        // Operands.
        Object operand1 = execute(operandNode1);
        Object operand2 = execute(operandNode2);

        boolean integerMode = (operand1 instanceof Integer) &&
                              (operand2 instanceof Integer);

        // ====================
        // Arithmetic operators
        // ====================

        if (ARITH_OPS.contains(nodeType)) {
            if (integerMode) {
                int value1 = (Integer) operand1;
                int value2 = (Integer) operand2;

                // Integer operations.
                switch (nodeType) {
                    case ADD:      return value1 + value2;
                    case SUBTRACT: return value1 - value2;
                    case MULTIPLY: return value1 * value2;

                    case FLOAT_DIVIDE: {

                        // Check for division by zero.
                        if (value2 != 0) {
                            return ((float) value1)/((float) value2);
                        }
                        else {
```

```
                    errorHandler.flag(node, DIVISION_BY_ZERO, this);
                    return 0;
                }
            }

            case INTEGER_DIVIDE: {

                // Check for division by zero.
                if (value2 != 0) {
                    return value1/value2;
                }
                else {
                    errorHandler.flag(node, DIVISION_BY_ZERO, this);
                    return 0;
                }
            }

            case MOD:  {

                // Check for division by zero.
                if (value2 != 0) {
                    return value1%value2;
                }
                else {
                    errorHandler.flag(node, DIVISION_BY_ZERO, this);
                    return 0;
                }
            }
        }
    }
    else {
        float value1 = operand1 instanceof Integer
                           ? (Integer) operand1 : (Float) operand1;
        float value2 = operand2 instanceof Integer
                           ? (Integer) operand2 : (Float) operand2;

        // Float operations.
        switch (nodeType) {
            case ADD:      return value1 + value2;
            case SUBTRACT: return value1 - value2;
            case MULTIPLY: return value1 * value2;

            case FLOAT_DIVIDE: {

                // Check for division by zero.
                if (value2 != 0.0f) {
                    return value1/value2;
                }
                else {
                    errorHandler.flag(node, DIVISION_BY_ZERO, this);
                    return 0.0f;
                }
            }
        }
    }
}

// ==========
// AND and OR
// ==========

else if ((nodeType == AND) || (nodeType == OR)) {
    boolean value1 = (Boolean) operand1;
    boolean value2 = (Boolean) operand2;

    switch (nodeType) {
        case AND: return value1 && value2;
        case OR:  return value1 || value2;
    }
}

// ====================
// Relational operators
// ====================

else if (integerMode) {
    int value1 = (Integer) operand1;
    int value2 = (Integer) operand2;

    // Integer operands.
    switch (nodeType) {
        case EQ: return value1 == value2;
        case NE: return value1 != value2;
        case LT: return value1 <  value2;
        case LE: return value1 <= value2;
        case GT: return value1 >  value2;
        case GE: return value1 >= value2;
    }
}
else {
    float value1 = operand1 instanceof Integer
                       ? (Integer) operand1 : (Float) operand1;
    float value2 = operand2 instanceof Integer
                       ? (Integer) operand2 : (Float) operand2;

    // Float operands.
    switch (nodeType) {
        case EQ: return value1 == value2;
        case NE: return value1 != value2;
        case LT: return value1 <  value2;
        case LE: return value1 <= value2;
        case GT: return value1 >  value2;
        case GE: return value1 >= value2;
    }
}

return 0;  // should never get here
}
```

Integer Arithmetic Operations

If the node that was passed in is an arithmetic operator node and local variable `integerMode` is true, then method `executeBinaryOperator()` performs integer arithmetic. It performs float arithmetic for a `FLOAT_DIVIDE` operator node.

For `ADD`, `SUBTRACT`, and `MULTIPLY` operator nodes, the method simply performs the operation on the two integer operand values and returns the integer result. For the `FLOAT_DIVIDE`, `INTEGER_DIVIDE`, and `MOD` operator nodes, the method checks that the value of the

second operand is not 0 before performing the division and flags a `DIVISION_BY_zero` runtime error if necessary. For `FLOAT_DIVIDE`, the method converts the integer operand values to `float` in order to do a float division.

Float Arithmetic Operations

If the node that was passed in is an arithmetic operator node but local variable `integerMode` is false, then method `executeBinaryOperator()` must perform float arithmetic. It first converts any integer operand value to float.

For `ADD`, `SUBTRACT`, `MULTIPLY`, and `FLOAT_DIVIDE` operator nodes, the method performs the operation on the two float operand values and returns the float result. For a `FLOAT_DIVIDE` operator node, the method checks that the value of the second operand is not 0 before performing the division and flags a `DIVISION_BY_ZERO` runtime error if necessary.

AND and OR Operations

If method `executeBinaryOperator()` received an `AND` or an `OR` operator node, then the method performs that operation on the two `Boolean` operand values and returns the result.

Relational Operations

If method `executeBinaryOperator()` received a relational operator node, it checks local variable `integerMode` to determine whether it must compare two integer operand values or two float operand values. In the latter case, the method first converts any integer operand value to float. It returns the `Boolean` result.

Design Note

Class `ExpressionExecutor` does not perform language-specific type checking. It relies on the language-specific parsers in the front end to do type checking and build a valid parse tree.

The class is also unaware of language-specific operator precedence rules. Again, it relies on the front end to build a proper parse tree that it can execute with a postorder traversal.

Program 6: Simple Interpreter I

You're finally ready for an end-to-end test of all the new code you've developed in this and the previous chapter. This test will parse a file `assignments.txt` that you saw in Listing 5-12, generate the parse tree, and then interpret the parse tree to execute the statements and produce printed results.

First, modify the main `Pascal` class and enable it to listen to the new runtime `ASSIGN` and `RUNTIME_ERROR` messages.

Listing 6-9 shows two new cases in `BackendMessageListener` inner class.

Listing 6-9: The **`ASSIGN`** and **`RUNTIME_ERROR`** cases in the inner class **`BackendMessageListener`** of the main **`Pascal`** class

```
case ASSIGN: {
    if (firstOutputMessage) {
        System.out.println("\n===== OUTPUT =====\n");
        firstOutputMessage = false;
    }

    Object body[] = (Object[]) message.getBody();
    int lineNumber = (Integer) body[0];
    String variableName = (String) body[1];
    Object value = body[2];

    System.out.printf(ASSIGN_FORMAT,
                      lineNumber, variableName, value);
    break;
}

case RUNTIME_ERROR: {
    Object body[] = (Object []) message.getBody();
    String errorMessage = (String) body[0];
    Integer lineNumber = (Integer) body[1];

    System.out.print("*** RUNTIME ERROR");
    if (lineNumber != null) {
        System.out.print(" AT LINE " +
                        String.format("%03d", lineNumber));
    }
    System.out.println(": " + errorMessage);
    break;
}
```

The `ASSIGN` case uses the string constant

```
private static final String ASSIGN_FORMAT =
    " >>> LINE %03d: %s = %s\n";
```

Again, assuming the class files are in the `classes` directory and the source file `assignments.txt` is in the current directory, a command line similar to

```
java -classpath classes Pascal execute assignments.txt
```

will parse and interpret the "program" in `assignments.txt`. Listing 6-10 shows the output. The interpreter caught and flagged a runtime division by zero error. The interpreter's summary messages from the back end now include the actual number of statements that it executed and the total execution time.

Listing 6-10: Parser and interpreter output

```
001 BEGIN
002     BEGIN {Temperature conversions.}
003         five  := -1 + 2 - 3 + 4 + 3;
004         ratio := five/9.0;
005
006         fahrenheit := 72;
007         centigrade := (fahrenheit - 32)*ratio;
008
009         centigrade := 25;
010         fahrenheit := centigrade/ratio + 32;
011
012         centigrade := 25;
013         fahrenheit := 32 + centigrade/ratio
014     END;
015
016     {Runtime division by zero error.}
017     dze := fahrenheit/(ratio - ratio);
018
019     BEGIN {Calculate a square root using Newton&apos;s method.}
020         number := 2;
021         root := number;
022         root := (number/root + root)/2;
023         root := (number/root + root)/2;
024         root := (number/root + root)/2;
025         root := (number/root + root)/2;
026         root := (number/root + root)/2;
027     END;
028
029     ch  := &apos;x&apos;;
030     str := &apos;hello, world&apos;
031 END.

                    31 source lines.
                     0 syntax errors.
                 0.05 seconds total parsing time.


===== OUTPUT =====

>>> LINE 003: five = 5
>>> LINE 004: ratio = 0.5555556
>>> LINE 006: fahrenheit = 72
>>> LINE 007: centigrade = 22.222223
>>> LINE 009: centigrade = 25
>>> LINE 010: fahrenheit = 77.0
>>> LINE 012: centigrade = 25
>>> LINE 013: fahrenheit = 77.0
*** RUNTIME ERROR AT LINE 017: Division by zero
>>> LINE 017: dze = 0.0
>>> LINE 020: number = 2
>>> LINE 021: root = 2
>>> LINE 022: root = 1.5
>>> LINE 023: root = 1.4166667
>>> LINE 024: root = 1.4142157
>>> LINE 025: root = 1.4142135
>>> LINE 026: root = 1.4142135
>>> LINE 029: ch = x
>>> LINE 030: str = hello, world

                    18 statements executed.
                     1 runtime errors.
                 0.02 seconds total execution time.
```

Design Note

The Chapter 6 Hacks

This chapter relies on several temporary "hacks" that allowed you to make progress on your rudimentary interpreter:

1. All variables are scalars (not records or arrays) but otherwise have no declared types. You haven't yet parsed variable declarations, the subject of Chapter 9.

2. The parser considers each variable to be "declared" the first time it encounters the variable as the target of an assignment statement and enters the variable into the symbol table at that time. You'll also fix this in Chapter 9.

3. You still have only one symbol table in the symbol table stack. You'll need multiple tables in Chapter 9 and again in Chapter 11.

4. The back end executors assume that the front end has done type checking and that the intermediate code is valid. But since you haven't yet parsed variable declarations, there was no type checking. You'll implement type checking in Chapter 10.

5. At runtime, each assignment statement stores the expression value into the target variable's symbol table entry. This won't work for a language like Pascal that allows recursive procedure and function calls. You'll implement runtime memory management in Chapter 12.

You'll remove these hacks as you move forward.

You have successfully written the beginnings of a *two-pass interpreter*. In the first pass, the front end parses the source and generates the intermediate code. In the second pass, the back end interprets the intermediate code to execute the source program's statements and expressions.

In the next chapter, you'll parse and generate intermediate code for Pascal control statements.

Chapter 7

Parsing Control Statements

In Chapter 5, you developed parsers in the front end to parse and generate intermediate code for Pascal assignment and composite statements and for expressions, and in Chapter 6, you developed executors in the interpreter back end to execute the statements and expressions. In this chapter you develop parsers for Pascal's control statements. Executors for the control statements will wait until the next chapter.

Goals and Approach

The goals for this chapter are:

- Parsers in the front end for Pascal control statements `WHILE`, `REPEAT`, `FOR`, `IF`, and `CASE`.
- Flexible, language-independent intermediate code generated by the parsers to represent these constructs.
- Reliable error recovery to ensure that the parsers can continue to work despite bad syntax errors in the source program.
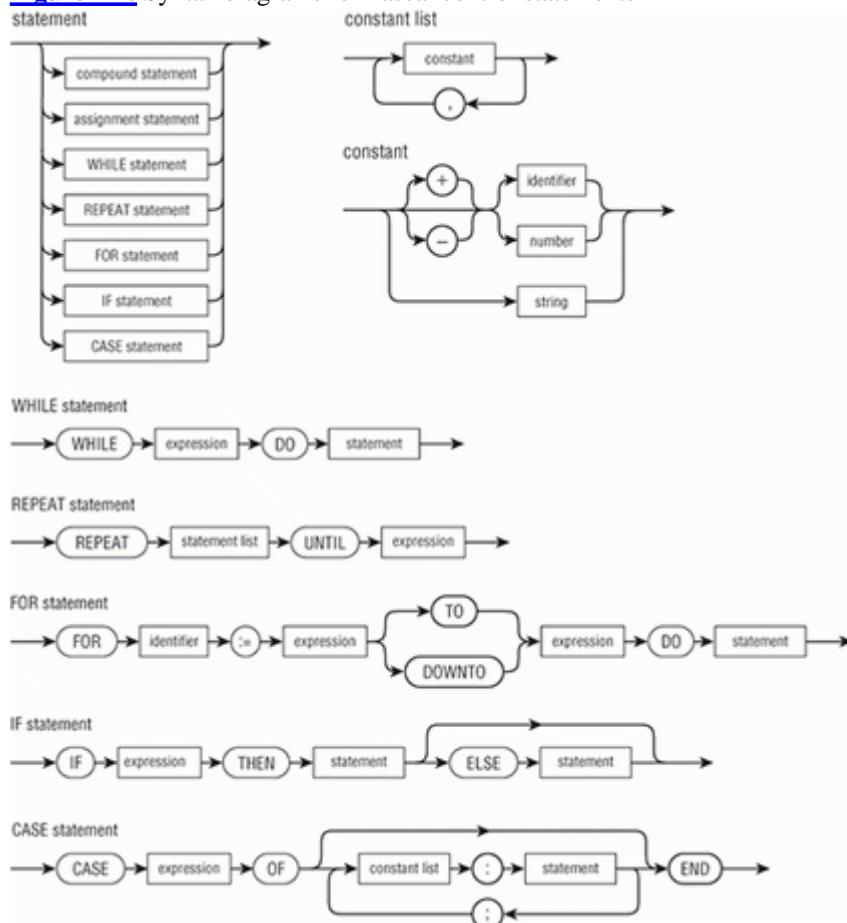
Our approach is the same as before. You'll start with syntax diagrams that will guide the development of new parser subclasses for each of the statements. To generate the parse trees, you'll code to the intermediate code interfaces that you developed in Chapter 5. You'll also extend the syntax checker utility program from that chapter to parse the control statements and verify the new code.

Syntax Diagrams

Figure 7-1 shows the syntax diagrams for Pascal control statements.[1]

[1] You won't handle Pascal's `goto` control statement in this book.

Figure 7-1: Syntax diagrams for Pascal control statements

As they have before, these diagrams will guide the development of our parser subclasses.

Error Recovery

Chapter 3 mentioned that syntax error handling in the front end is a three-step process: *detection*, *flagging*, and *recovery*. Up until now, the parser subclasses used very rudimentary recovery. Whenever a parser detected an error, it flagged the erroneous token and attempted to move forward. If the error was a missing token, the parser assumed the token was present. If the token was unexpected, the parser consumed it and hoped that the next token would make sense.

What are a parser's options for error recovery?

- It can simply terminate after encountering a syntax error. In the worst cases, it can hang by getting stuck on a token that's never consumed or even crash. In other words, there is no error recovery at all. This option is easy for the compiler writer but extremely annoying for programmers attempting to use the compiler.
- It can become hopelessly lost but still attempt to parse the rest of the source program while spitting out sequences of irrelevant error messages. There's no error recovery here, either, but the compiler writer doesn't want to admit it.
- It can skip tokens after the erroneous one until it finds a token it recognizes and safely resume syntax checking the rest of the source program.

Clearly, the first two options are undesirable. To implement the third option, the parser must "synchronize" itself frequently at tokens that it expects. Whenever there is a syntax error, the parser must find the next token in the source program where it can reliably resume syntax checking. Ideally, it can find such a token as soon after the error as possible.

Listing 7-1 shows the new `synchronize()` method you need to add to the Pascal parser class `PascalParserTD` in package `frontend.pascal`. This method will be used by the parser subclasses to remain properly synchronized.

Listing 7-1: The new `synchronize()` method in class `PascalParserTD`

```
/**
 * Synchronize the parser.
 * @param syncSet the set of token types for synchronizing the parser.
 * @return the token where the parser has synchronized.
 * @throws Exception if an error occurred.
 */
public Token synchronize(EnumSet syncSet)
    throws Exception
{
    Token token = currentToken();

    // If the current token is not in the synchronization set,
    // then it is unexpected and the parser must recover.
    if (!syncSet.contains(token.getType())) {

        // Flag the unexpected token.
        errorHandler.flag(token, UNEXPECTED_TOKEN, this);

        // Recover by skipping tokens that are not
        // in the synchronization set.
        do {
            token = nextToken();
        } while (!(token instanceof EofToken) &&
                  !syncSet.contains(token.getType()));
    }

    return token;
}
```

The method's callers pass in a *synchronization set* of Pascal token types. The method checks if the current token's type is in the set. If it is, there is no syntax error and the method immediately returns the token. If not, the method flags an UNEXPECTED_TOKEN error and recovers by skipping the subsequent tokens until it reaches the first one whose type is in the set. In either case, the parser is synchronized at the returned token.

Design Note

Error recovery with top-down parsers can be more art than science, and it usually takes a compiler writer several tries to get it right for each source language construct. The parsers in this book use a more simplistic than sophisticated approach. The key for method `synchronize()` is the content of its synchronization set argument, which determines how many tokens the parser skips after a syntax error before it starts parsing again. Skipping too much, perhaps the rest of the source program, is considered to be a "panic mode" recovery.

Program 7: Syntax Checker II

There won't be any changes from Chapter 6 to the main class `Pascal`. Of course, the syntax-checking capabilities of the program as a whole will increase with the addition of the new parser subclasses for the control statements.
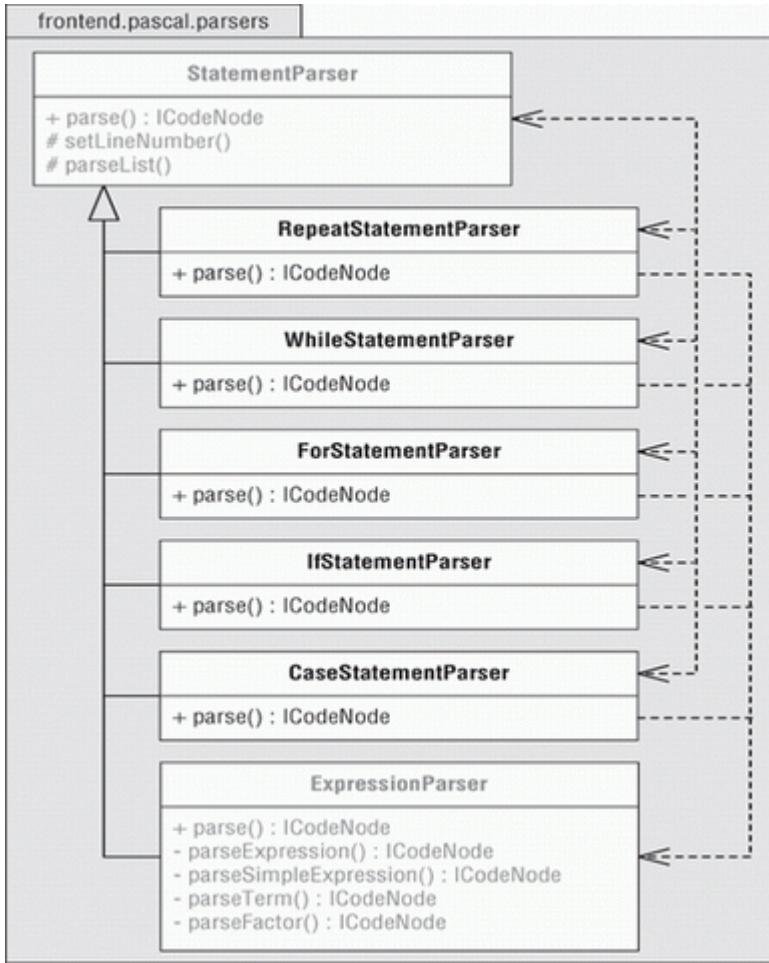
You'll run the program several times this chapter on various source files, including ones containing syntax errors, to verify the new parser subclasses. But since you haven't yet written the corresponding executor subclasses in the interpreter back end, you'll run the program as a compiler in this chapter to avoid getting UNIMPLEMENTED_FEATURE runtime errors.

Control Statement Parsers

The UML diagram in
Figure 7-2 adds to the diagram in Figure 5-7.

Figure 7-2: The statement parser subclasses for Pascal control statements

frontend.pascal.parsers

Each control statement parser is a subclass of `StatementParser` which in turn depends on each one of them. Furthermore, because each control statement contains a nested statement, each control statement parser depends on `StatementParser`. Hence the dependency lines with arrowheads at both ends. Each control statement contains an expression, so each control statement parser also depends on `ExpressionParser`. A `FOR` statement contains an embedded assignment statement, and so `ForStatementParser` also depends on `AssignmentStatementParser`, which is not shown in this diagram.

First update the Pascal parser subclass `StatementParser` and its subclass `AssignmentStatementParser`, both of which you developed in Chapter 5. Listing 7-2 shows the new version of the `parse()` method of `StatementParser`.

Listing 7-2: Method `parse()` of class `StatementParser`

```
// Synchronization set for starting a statement.
protected static final EnumSet<PascalTokenType> STMT_START_SET =
    EnumSet.of(BEGIN, CASE, FOR, PascalTokenType.IF, REPEAT, WHILE,
               IDENTIFIER, SEMICOLON);

// Synchronization set for following a statement.
protected static final EnumSet<PascalTokenType> STMT_FOLLOW_SET =
    EnumSet.of(SEMICOLON, END, ELSE, UNTIL, DOT);

/**
 * Parse a statement.
 * To be overridden by the specialized statement parser subclasses.
 * @param token the initial token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token)
    throws Exception
{
    ICodeNode statementNode = null;

    switch ((PascalTokenType) token.getType()) {

        case BEGIN: {
            CompoundStatementParser compoundParser =
                new CompoundStatementParser(this);
            statementNode = compoundParser.parse(token);
            break;
        }

        // An assignment statement begins with a variable's identifier.
        case IDENTIFIER: {
            AssignmentStatementParser assignmentParser =
                new AssignmentStatementParser(this);
            statementNode = assignmentParser.parse(token);
            break;
        }

        case REPEAT: {
            RepeatStatementParser repeatParser =
                new RepeatStatementParser(this);
            statementNode = repeatParser.parse(token);
            break;
        }

        case WHILE: {
            WhileStatementParser whileParser =
                new WhileStatementParser(this);
```

```
                statementNode = whileParser.parse(token);
                break;
            }

            case FOR: {
                ForStatementParser forParser = new ForStatementParser(this);
                statementNode = forParser.parse(token);
                break;
            }

            case IF: {
                IfStatementParser ifParser = new IfStatementParser(this);
                statementNode = ifParser.parse(token);
                break;
            }

            case CASE: {
                CaseStatementParser caseParser = new CaseStatementParser(this);
                statementNode = caseParser.parse(token);
                break;
            }

            default: {
                statementNode = ICodeFactory.createICodeNode(NO_OP);
                break;
            }
        }

        // Set the current line number as an attribute.
        setLineNumber(statementNode, token);

        return statementNode;
    }
```

The synchronization set `STMT_START_SET` contains the token types that can start a statement and the synchronization set `STMT_FOLLOW_SET` contains the token types that can follow a statement. `STMT_START_SET` includes the semicolon to handle the empty statement. The `parse()` method now can handle Pascal control statements.

Listing 7-3 shows the updated version of method `parseList()` of `StatementParser`. It creates the synchronization set `terminatorSet` by cloning `STMT_START_SET` and adding the `terminator` token type. The end of the `while` loop calls `synchronize (terminatorSet)` to synchronize at the terminator token or at the start of the next statement.

Listing 7-3: Method `parseList()` of class `StatementParser`

```
    /**
     * Parse a statement list.
     * @param token the curent token.
     * @param parentNode the parent node of the statement list.
     * @param terminator the token type of the node that terminates the list.
     * @param errorCode the error code if the terminator token is missing.
     * @throws Exception if an error occurred.
     */
    protected void parseList(Token token, ICodeNode parentNode,
                             PascalTokenType terminator,
                             PascalErrorCode errorCode)
        throws Exception
    {
        // Synchronization set for the terminator.
        EnumSet<PascalTokenType> terminatorSet = STMT_START_SET.clone();
        terminatorSet.add(terminator);

        // Loop to parse each statement until the END token
        // or the end of the source file.
        while (!(token instanceof EofToken) &&
               (token.getType() != terminator)) {

            // Parse a statement.  The parent node adopts the statement node.
            ICodeNode statementNode = parse(token);
            parentNode.addChild(statementNode);

            token = currentToken();
            TokenType tokenType = token.getType();

            // Look for the semicolon between statements.
            if (tokenType == SEMICOLON) {
                token = nextToken();  // consume the ;
            }

            // If at the start of the next statement, then missing a semicolon.
            else if (STMT_START_SET.contains(tokenType)) {
                errorHandler.flag(token, MISSING_SEMICOLON, this);
            }

            // Synchronize at the start of the next statement
            // or at the terminator.
            token = synchronize(terminatorSet);
        }

        // Look for the terminator token.
        if (token.getType() == terminator) {
            token = nextToken();  // consume the terminator token
        }
        else {
            errorHandler.flag(token, errorCode, this);
        }
    }
```

Listing 7-4 shows the updated `parse()` method of the statement parser subclass `AssignmentStatementParser`.

Listing 7-4: Method `parse()` of class `AssignmentStatementParser`

```
    // Synchronization set for the := token.
    private static final EnumSet<PascalTokenType> COLON_EQUALS_SET =
        ExpressionParser.EXPR_START_SET.clone();
    static {
        COLON_EQUALS_SET.add(COLON_EQUALS);
        COLON_EQUALS_SET.addAll(StatementParser.STMT_FOLLOW_SET);
    }

    /**
     * Parse an assignment statement.
     * @param token the initial token.
     * @return the root node of the generated parse tree.
     * @throws Exception if an error occurred.
     */
```

```
public ICodeNode parse(Token token)
    throws Exception
{
    // Create the ASSIGN node.
    ICodeNode assignNode = ICodeFactory.createICodeNode(ASSIGN);

    // Look up the target identifer in the symbol table stack.
    // Enter the identifier into the table if it&apos;s not found.
    String targetName = token.getText().toLowerCase();
    SymTabEntry targetId = symTabStack.lookup(targetName);
    if (targetId == null) {
        targetId = symTabStack.enterLocal(targetName);
    }
    targetId.appendLineNumber(token.getLineNumber());

    token = nextToken();  // consume the identifier token

    // Create the variable node and set its name attribute.
    ICodeNode variableNode = ICodeFactory.createICodeNode(VARIABLE);
    variableNode.setAttribute(ID, targetId);

    // The ASSIGN node adopts the variable node as its first child.
    assignNode.addChild(variableNode);

    // Synchronize on the := token.
    token = synchronize(COLON_EQUALS_SET);
    if (token.getType() == COLON_EQUALS) {
        token = nextToken();  // consume the :=
    }
    else {
        errorHandler.flag(token, MISSING_COLON_EQUALS, this);
    }

    // Parse the expression.  The ASSIGN node adopts the expression&apos;s
    // node as its second child.
    ExpressionParser expressionParser = new ExpressionParser(this);
    assignNode.addChild(expressionParser.parse(token));

    return assignNode;
}
```

After parsing the target variable, method `parse()` calls `synchronize(COLON_EQUALS_SET)` to synchronize itself at the `:=` token. The synchronization set is a clone of `EXPR_START_SET` with the addition of the `COLON_EQUALS` token type. To help prevent a bad panic mode recovery, this set and other similar sets also include all the token types in `StatementParser.STMT_FOLLOW_SET`. If the parser can't find the `:=` token, it will synchronize at the start of the expression, or in a worse situation, at the first token that can follow the assignment statement.

Class `ExpressionParser` defines the synchronization set `EXPR_START_SET`:

```
// Synchronization set for starting an expression.
static final EnumSet<PascalTokenType> EXPR_START_SET =
    EnumSet.of(PLUS, MINUS, IDENTIFIER, INTEGER, REAL, STRING,
               PascalTokenType.NOT, LEFT_PAREN);
```

Parsing Pascal Control Statements

Now you're ready to parse Pascal control statements.

Parsing the **REPEAT** Statement

The statement parser subclass `RepeatStatementParser` parses a Pascal `REPEAT` statement and generates its parse tree. For the statement

```
REPEAT
    j := i;
    k := i
UNTIL i <= j
```

the `parse()` method generates the parse tree shown in .

Figure 7-3: Parse tree for a `REPEAT` statement

The `LOOP` node can have any number of children that are statement subtrees. At least one child should be a `TEST` node whose only child is a relational expression subtree. At runtime, the loop exits if the expression evaluates to true. A `TEST` node can be any one of the `LOOP` node's children, so the exit test can occur at the start of the loop, at the end of the loop, or somewhere in between. For a Pascal `REPEAT` statement, the `TEST` node is the `LOOP` node's last child and therefore, the exit test is at the end of the loop.

shows the `parse()` method of class `RepeatStatementParser`.

Listing 7-5: Method `parse()` of class `RepeatStatementParser`

```
    /**
     * Parse a REPEAT statement.
     * @param token the initial token.
     * @return the root node of the generated parse tree.
     * @throws Exception if an error occurred.
     */
    public ICodeNode parse(Token token)
        throws Exception
    {
        token = nextToken();  // consume the REPEAT

        // Create the LOOP and TEST nodes.
        ICodeNode loopNode = ICodeFactory.createICodeNode(LOOP);
        ICodeNode testNode = ICodeFactory.createICodeNode(TEST);

        // Parse the statement list terminated by the UNTIL token.
        // The LOOP node is the parent of the statement subtrees.
        StatementParser statementParser = new StatementParser(this);
        statementParser.parseList(token, loopNode, UNTIL, MISSING_UNTIL);
        token = currentToken();

        // Parse the expression.
        // The TEST node adopts the expression subtree as its only child.
        // The LOOP node adopts the TEST node.
        ExpressionParser expressionParser = new ExpressionParser(this);
        testNode.addChild(expressionParser.parse(token));
        loopNode.addChild(testNode);

        return loopNode;
    }
```

The `parse()` method creates the LOOP and TEST nodes and builds the parse tree shown in <u>Figure 7-3</u>. The method calls `statementParser.parseList()` to parse a statement list that is terminated by the UNTIL token. The LOOP node becomes the parent of the statement subtrees.

Assuming the class files are in the `classes` directory and the source file `loops.txt` are in the current directory, a command line similar to[2]

---

[2] Remember that you're running the program as a compiler in this chapter because you haven't yet written all the statement executors in the interpreter back end.

```
java -classpath classes Pascal compile -i repeat.txt
```

will run the syntax checker to parse the REPEAT statements in source file `repeat.txt` and generate the output shown in <u>Listing 7-6</u>.

<u>Listing 7-6:</u> Output from "compiling" REPEAT statements

```
001 BEGIN {REPEAT statements.}
002    i := 0;
003
004    REPEAT
005        j := i;
006        k := i
007    UNTIL i <= j;
008
009    BEGIN {Calculate a square root using Newton&apos;s method.}
010        number := 4;
011        root := number;
012
013        REPEAT
014            partial := number/root + root;
015            root := partial/2
016        UNTIL root*root - number < 0.000001
017    END
018 END.

              18 source lines.
               0 syntax errors.
             0.05 seconds total parsing time.

===== INTERMEDIATE CODE =====

<COMPOUND line="1">
    <ASSIGN line="2">
        <VARIABLE id="i" level="0" />
        <INTEGER_CONSTANT value="0" />
    </ASSIGN>
    <LOOP line="4">
        <ASSIGN line="5">
            <VARIABLE id="j" level="0" />
            <VARIABLE id="i" level="0" />
        </ASSIGN>
        <ASSIGN line="6">
            <VARIABLE id="k" level="0" />
            <VARIABLE id="i" level="0" />
        </ASSIGN>
        <TEST>
            <LE>
                <VARIABLE id="i" level="0" />
                <VARIABLE id="j" level="0" />
            </LE>
        </TEST>
    </LOOP>
    <COMPOUND line="9">
        <ASSIGN line="10">
            <VARIABLE id="number" level="0" />
            <INTEGER_CONSTANT value="4" />
        </ASSIGN>
        <ASSIGN line="11">
            <VARIABLE id="root" level="0" />
            <VARIABLE id="number" level="0" />
        </ASSIGN>
        <LOOP line="13">
            <ASSIGN line="14">
                <VARIABLE id="partial" level="0" />
                <ADD>
                    <FLOAT_DIVIDE>
                        <VARIABLE id="number" level="0" />
                        <VARIABLE id="root" level="0" />
```

```
                        </FLOAT_DIVIDE>
                        <VARIABLE id="root" level="0" />
                    </ADD>
                </ASSIGN>
                <ASSIGN line="15">
                    <VARIABLE id="root" level="0" />
                    <FLOAT_DIVIDE>
                        <VARIABLE id="partial" level="0" />
                        <INTEGER_CONSTANT value="2" />
                    </FLOAT_DIVIDE>
                </ASSIGN>
                <TEST>
                    <LT>
                        <SUBTRACT>
                            <MULTIPLY>
                                <VARIABLE id="root" level="0" />
                                <VARIABLE id="root" level="0" />
                            </MULTIPLY>
                            <VARIABLE id="number" level="0" />
                        </SUBTRACT>
                        <REAL_CONSTANT value="1.0E-6" />
                    </LT>
                </TEST>
            </LOOP>
        </COMPOUND>
</COMPOUND>

            0 instructions generated.
            0.00 seconds total code generation time.
```

Listing 7-7 shows REPEAT statement syntax error handling.

Listing 7-7: REPEAT **statement syntax error handling**

```
001 BEGIN {REPEAT syntax errors}
002     REPEAT UNTIL five := 5;
                        ^
*** Undefined identifier [at "five"]
                             ^
*** Unexpected token [at ":="]
003     REPEAT ratio := 9 UNTIL;
                              ^
*** Unexpected token [at ";"]
004 END.

            4 source lines.
            3 syntax errors.
            0.05 seconds total parsing time.
```
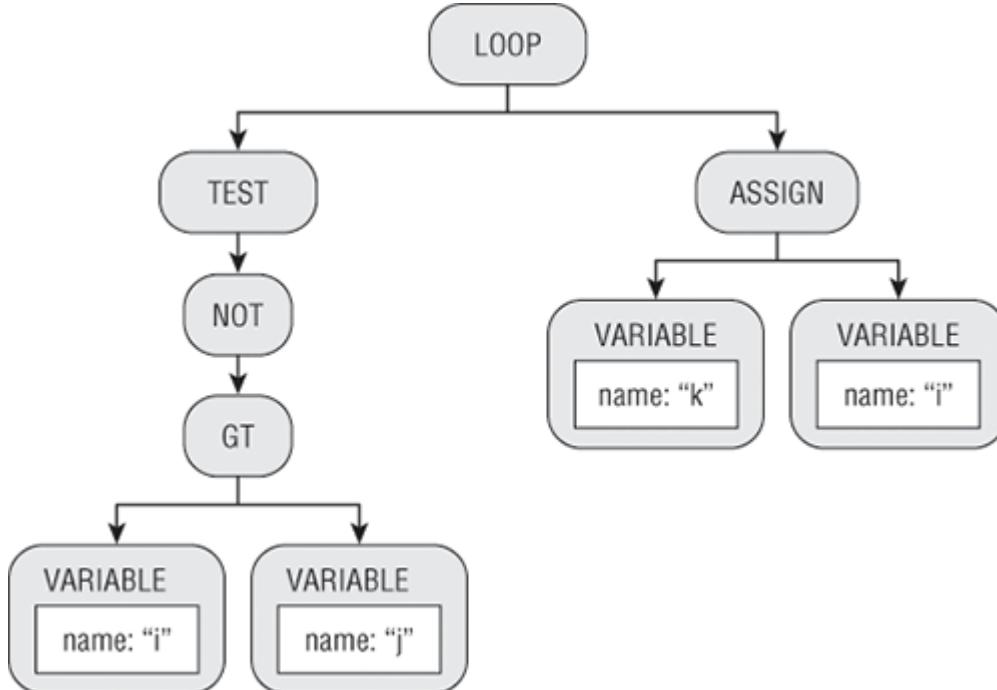
Parsing the WHILE Statement

The statement parser subclass WhileStatementParser parses a Pascal WHILE statement and generates its parse tree. For the statement

```
WHILE i > j DO k := i
```

the parse() method generates the parse tree shown in Figure 7-4.

Figure 7-4: Parse tree for a WHILE statement

For a Pascal WHILE statement, the LOOP node's first child is the TEST node and the second child is the subtree of the nested statement. Therefore, at runtime, the exit test occurs at the start of the loop. But because a WHILE loop exits when the test expression is false, the parent of the relational expression subtree is a generated NOT node.

Listing 7-8 shows the parse() method of class WhileStatementParser.

Listing 7-8: Method parse() **of class** WhileStatementParser

```java
    // Synchronization set for DO.
    private static final EnumSet<PascalTokenType> DO_SET =
        StatementParser.STMT_START_SET.clone();
    static {
        DO_SET.add(DO);
        DO_SET.addAll(StatementParser.STMT_FOLLOW_SET);
    }

    /**
     * Parse a WHILE statement.
     * @param token the initial token.
     * @return the root node of the generated parse tree.
     * @throws Exception if an error occurred.
     */
    public ICodeNode parse(Token token)
        throws Exception
    {
        token = nextToken();  // consume the WHILE

        // Create LOOP, TEST, and NOT nodes.
        ICodeNode loopNode = ICodeFactory.createICodeNode(LOOP);
        ICodeNode breakNode = ICodeFactory.createICodeNode(TEST);
        ICodeNode notNode = ICodeFactory.createICodeNode(ICodeNodeTypeImpl.NOT);

        // The LOOP node adopts the TEST node as its first child.
        // The TEST node adopts the NOT node as its only child.
        loopNode.addChild(breakNode);
        breakNode.addChild(notNode);

        // Parse the expression.
        // The NOT node adopts the expression subtree as its only child.
        ExpressionParser expressionParser = new ExpressionParser(this);
        notNode.addChild(expressionParser.parse(token));

        // Synchronize at the DO.
        token = synchronize(DO_SET);
        if (token.getType() == DO) {
            token = nextToken();  // consume the DO
        }
        else {
            errorHandler.flag(token, MISSING_DO, this);
        }

        // Parse the statement.
        // The LOOP node adopts the statement subtree as its second child.
        StatementParser statementParser = new StatementParser(this);
        loopNode.addChild(statementParser.parse(token));

        return loopNode;
    }
```

The `parse()` method creates the LOOP, TEST, and NOT nodes and builds the parse tree shown in . It uses the synchronization set DO_SET to synchronize itself at the DO token.

shows syntax checker output from compiling WHILE statements.

Listing 7-9: Output from "compiling" WHILE statements

```
001 BEGIN {WHILE statements}
002     i := 0;  j := 0;
003
004     WHILE i > j DO k := i;
005
006     BEGIN {Calculate a square root using Newton&apos;s method.}
007         number := 2;
008         root := number;
009
010         WHILE root*root - number > 0.000001 DO BEGIN
011             root := (number/root + root)/2
012         END
013     END;
014 END.

                14 source lines.
                 0 syntax errors.
                0.05 seconds total parsing time.

===== INTERMEDIATE CODE =====

<COMPOUND line="1">
    <ASSIGN line="2">
        <VARIABLE id="i" level="0" />
        <INTEGER_CONSTANT value="0" />
    </ASSIGN>
    <ASSIGN line="2">
        <VARIABLE id="j" level="0" />
        <INTEGER_CONSTANT value="0" />
    </ASSIGN>
    <LOOP line="4">
        <TEST>
            <NOT>
                <GT>
                    <VARIABLE id="i" level="0" />
                    <VARIABLE id="j" level="0" />
                </GT>
            </NOT>
        </TEST>
        <ASSIGN line="4">
            <VARIABLE id="k" level="0" />
            <VARIABLE id="i" level="0" />
        </ASSIGN>
    </LOOP>
    <COMPOUND line="6">
        <ASSIGN line="7">
            <VARIABLE id="number" level="0" />
            <INTEGER_CONSTANT value="2" />
        </ASSIGN>
        <ASSIGN line="8">
            <VARIABLE id="root" level="0" />
            <VARIABLE id="number" level="0" />
        </ASSIGN>
        <LOOP line="10">
            <TEST>
                <NOT>
```
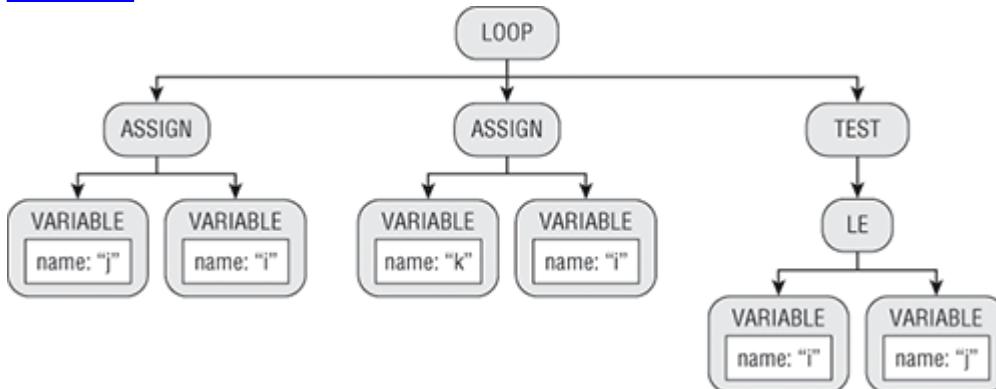
```
            <GT>
                <SUBTRACT>
                    <MULTIPLY>
                        <VARIABLE id="root" level="0" />
                        <VARIABLE id="root" level="0" />
                    </MULTIPLY>
                    <VARIABLE id="number" level="0" />
                </SUBTRACT>
                <REAL_CONSTANT value="1.0E-6" />
            </GT>
        </NOT>
    </TEST>
    <COMPOUND line="10">
        <ASSIGN line="11">
            <VARIABLE id="root" level="0" />
            <FLOAT_DIVIDE>
                <ADD>
                    <FLOAT_DIVIDE>
                        <VARIABLE id="number" level="0" />
                        <VARIABLE id="root" level="0" />
                    </FLOAT_DIVIDE>
                    <VARIABLE id="root" level="0" />
                </ADD>
                <INTEGER_CONSTANT value="2" />
            </FLOAT_DIVIDE>
        </ASSIGN>
    </COMPOUND>
    </LOOP>
    </COMPOUND>
</COMPOUND>

            0 instructions generated.
            0.00 seconds total code generation time.
```

Listing 7-10 shows WHILE statement syntax error handling.

Listing 7-10: WHILE statement syntax error handling

```
001 BEGIN {WHILE syntax errors}
002     WHILE DO five := 5;
                  ^
*** Unexpected token [at "DO"]
003     WHILE five = 5 five := 5 UNTIL five := 9;
                        ^
*** Missing DO [at "five"]
                                  ^
*** Unexpected token [at "UNTIL"]
004 END.

            4 source lines.
            3 syntax errors.
            0.05 seconds total parsing time.
```
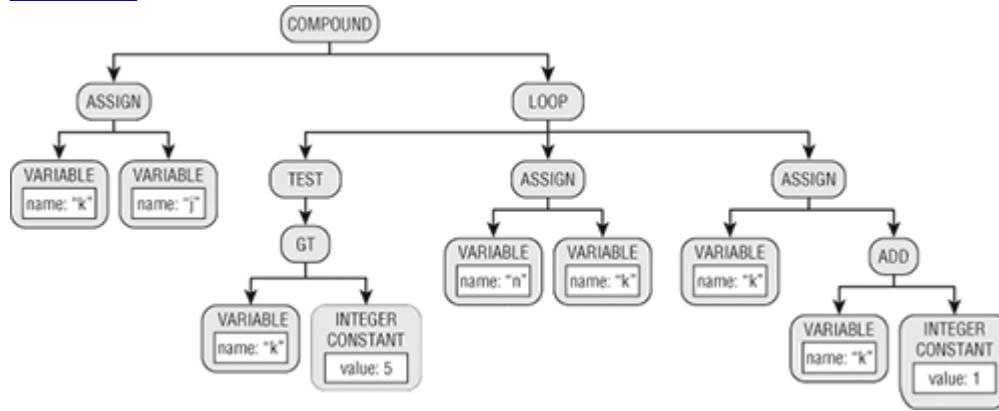
Parsing the FOR Statement

The Pascal FOR statement is a bit of a challenge. The parse() method of the statement parser subclass ForStatementParser parses the statement

```
FOR k := j TO 5 DO n := k
```

and generates the parse tree shown in Figure 7-5.

Figure 7-5: Parse tree for a FOR statement

The root of the parse tree is a COMPOUND node. The COMPOUND node's first child is the subtree of the embedded assignment which initializes the control variable. The second child is a LOOP node.

The first child of the LOOP node is a TEST node. The TEST node's child is either a GT or the LT relational expression subtree, depending on whether the FOR statement is TO or DOWNTO, which tests the control variable's value against the final value. The second child of the LOOP node is the subtree of the nested statement. The third child is either an ADD or a SUBTRACT arithmetic expression subtree, again depending on TO or DOWNTO, which increments or decrements the control variable's value by 1.

Listing 7-11 shows the parse() method of class ForStatementParser.

Listing 7-11: Method parse() of class ForStatementParser

```
    // Synchronization set for TO or DOWNTO.
    static final EnumSet<PascalTokenType> TO_DOWNTO_SET =
        ExpressionParser.EXPR_START_SET.clone();
    static {
        TO_DOWNTO_SET.add(TO);
```

```
        TO_DOWNTO_SET.add(DOWNTO);
        TO_DOWNTO_SET.addAll(StatementParser.STMT_FOLLOW_SET);
    }

    // Synchronization set for DO.
    private static final EnumSet<PascalTokenType> DO_SET =
        StatementParser.STMT_START_SET.clone();
    static {
        DO_SET.add(DO);
        DO_SET.addAll(StatementParser.STMT_FOLLOW_SET);
    }

    /**
     * Parse the FOR statement.
     * @param token the initial token.
     * @return the root node of the generated parse tree.
     * @throws Exception if an error occurred.
     */
    public ICodeNode parse(Token token)
        throws Exception
    {
        token = nextToken();  // consume the FOR
        Token targetToken = token;

        // Create the loop COMPOUND, LOOP, and TEST nodes.
        ICodeNode compoundNode = ICodeFactory.createICodeNode(COMPOUND);
        ICodeNode loopNode = ICodeFactory.createICodeNode(LOOP);
        ICodeNode testNode = ICodeFactory.createICodeNode(TEST);

        // Parse the embedded initial assignment.
        AssignmentStatementParser assignmentParser =
            new AssignmentStatementParser(this);
        ICodeNode initAssignNode = assignmentParser.parse(token);

        // Set the current line number attribute.
        setLineNumber(initAssignNode, targetToken);

        // The COMPOUND node adopts the initial ASSIGN and the LOOP nodes
        // as its first and second children.
        compoundNode.addChild(initAssignNode);
        compoundNode.addChild(loopNode);

        // Synchronize at the TO or DOWNTO.
        token = synchronize(TO_DOWNTO_SET);
        TokenType direction = token.getType();

        // Look for the TO or DOWNTO.
        if ((direction == TO) || (direction == DOWNTO)) {
            token = nextToken();  // consume the TO or DOWNTO
        }
        else {
            direction = TO;
            errorHandler.flag(token, MISSING_TO_DOWNTO, this);
        }

        // Create a relational operator node: GT for TO, or LT for DOWNTO.
        ICodeNode relOpNode = ICodeFactory.createICodeNode(direction == TO
                                                           ? GT : LT);

        // Copy the control VARIABLE node. The relational operator
        // node adopts the copied VARIABLE node as its first child.
        ICodeNode controlVarNode = initAssignNode.getChildren().get(0);
        relOpNode.addChild(controlVarNode.copy());

        // Parse the termination expression. The relational operator node
        // adopts the expression as its second child.
        ExpressionParser expressionParser = new ExpressionParser(this);
        relOpNode.addChild(expressionParser.parse(token));

        // The TEST node adopts the relational operator node as its only child.
        // The LOOP node adopts the TEST node as its first child.
        testNode.addChild(relOpNode);
        loopNode.addChild(testNode);

        // Synchronize at the DO.
        token = synchronize(DO_SET);
        if (token.getType() == DO) {
            token = nextToken();  // consume the DO
        }
        else {
            errorHandler.flag(token, MISSING_DO, this);
        }

        // Parse the nested statement. The LOOP node adopts the statement
        // node as its second child.
        StatementParser statementParser = new StatementParser(this);
        loopNode.addChild(statementParser.parse(token));

        // Create an assignment with a copy of the control variable
        // to advance the value of the variable.
        ICodeNode nextAssignNode = ICodeFactory.createICodeNode(ASSIGN);
        nextAssignNode.addChild(controlVarNode.copy());

        // Create the arithmetic operator node:
        // ADD for TO, or SUBTRACT for DOWNTO.
        ICodeNode arithOpNode = ICodeFactory.createICodeNode(direction == TO
                                                             ? ADD : SUBTRACT);

        // The operator node adopts a copy of the loop variable as its
        // first child and the value 1 as its second child.
        arithOpNode.addChild(controlVarNode.copy());
        ICodeNode oneNode = ICodeFactory.createICodeNode(INTEGER_CONSTANT);
        oneNode.setAttribute(VALUE, 1);
        arithOpNode.addChild(oneNode);

        // The next ASSIGN node adopts the arithmetic operator node as its
        // second child. The loop node adopts the next ASSIGN node as its
        // third child.
        nextAssignNode.addChild(arithOpNode);
        loopNode.addChild(nextAssignNode);

        // Set the current line number attribute.
        setLineNumber(nextAssignNode, targetToken);

        return compoundNode;
    }
```

The parse() method creates the COMPOUND, LOOP, and TEST nodes and builds the parse tree shown in . It uses the synchronization set TO_DOWNTO_SET to synchronize itself at the TO or DOWNTO token and the synchronization set DO_SET to synchronize itself at the DO token. The method sets the source line number for both the "initial" and the "next" ASSIGN nodes.

The method remembers whether it saw TO or DOWNTO. If it saw TO, it generates a GT relational operator node and an ADD arithmetic operator node. Otherwise, if it saw DOWNTO, it generates instead LT and SUBTRACT operator nodes.

Listing 7-12 shows syntax checker output from compiling FOR statements.

Listing 7-12: Output from "compiling" FOR statements

```
001 BEGIN {FOR statements}
002     j := 1;
003
004     FOR k := j TO 5 DO n := k;
005
006     FOR k := n DOWNTO 1 DO j := k;
007
008     FOR i := 1 TO 2 DO BEGIN
009         FOR j := 1 TO 3 DO BEGIN
010             k := i*j
011         END
012     END
013 END.

                13 source lines.
                 0 syntax errors.
                0.06 seconds total parsing time.


===== INTERMEDIATE CODE =====

<COMPOUND line="1">
    <ASSIGN line="2">
        <VARIABLE id="j" level="0" />
        <INTEGER_CONSTANT value="1" />
    </ASSIGN>
    <COMPOUND line="4">
        <ASSIGN line="4">
            <VARIABLE id="k" level="0" />
            <VARIABLE id="j" level="0" />
        </ASSIGN>
        <LOOP>
            <TEST>
                <GT>
                    <VARIABLE id="k" level="0" />
                    <INTEGER_CONSTANT value="5" />
                </GT>
            </TEST>
            <ASSIGN line="4">
                <VARIABLE id="n" level="0" />
                <VARIABLE id="k" level="0" />
            </ASSIGN>
            <ASSIGN line="4">
                <VARIABLE id="k" level="0" />
                <ADD>
                    <VARIABLE id="k" level="0" />
                    <INTEGER_CONSTANT value="1" />
                </ADD>
            </ASSIGN>
        </LOOP>
    </COMPOUND>
    <COMPOUND line="6">
        <ASSIGN line="6">
            <VARIABLE id="k" level="0" />
            <VARIABLE id="n" level="0" />
        </ASSIGN>
        <LOOP>
            <TEST>
                <LT>
                    <VARIABLE id="k" level="0" />
                    <INTEGER_CONSTANT value="1" />
                </LT>
            </TEST>
            <ASSIGN line="6">
                <VARIABLE id="j" level="0" />
                <VARIABLE id="k" level="0" />
            </ASSIGN>
            <ASSIGN line="6">
                <VARIABLE id="k" level="0" />
                <SUBTRACT>
                    <VARIABLE id="k" level="0" />
                    <INTEGER_CONSTANT value="1" />
                </SUBTRACT>
            </ASSIGN>
        </LOOP>
    </COMPOUND>
    <COMPOUND line="8">
        <ASSIGN line="8">
            <VARIABLE id="i" level="0" />
            <INTEGER_CONSTANT value="1" />
        </ASSIGN>
        <LOOP>
            <TEST>
                <GT>
                    <VARIABLE id="i" level="0" />
                    <INTEGER_CONSTANT value="2" />
                </GT>
            </TEST>
            <COMPOUND line="8">
                <COMPOUND line="9">
                    <ASSIGN line="9">
                        <VARIABLE id="j" level="0" />
                        <INTEGER_CONSTANT value="1" />
                    </ASSIGN>
                    <LOOP>
                        <TEST>
                            <GT>
                                <VARIABLE id="j" level="0" />
                                <INTEGER_CONSTANT value="3" />
                            </GT>
                        </TEST>
                        <COMPOUND line="9">
                            <ASSIGN line="10">
```

```
                    <VARIABLE id="k" level="0" />
                    <MULTIPLY>
                        <VARIABLE id="i" level="0" />
                        <VARIABLE id="j" level="0" />
                    </MULTIPLY>
                </ASSIGN>
            </COMPOUND>
            <ASSIGN line="9">
                <VARIABLE id="j" level="0" />
                <ADD>
                    <VARIABLE id="j" level="0" />
                    <INTEGER_CONSTANT value="1" />
                </ADD>
            </ASSIGN>
        </LOOP>
      </COMPOUND>
    </COMPOUND>
    <ASSIGN line="8">
      <VARIABLE id="i" level="0" />
      <ADD>
          <VARIABLE id="i" level="0" />
          <INTEGER_CONSTANT value="1" />
      </ADD>
    </ASSIGN>
  </LOOP>
  </COMPOUND>
</COMPOUND>

          0 instructions generated.
          0.00 seconds total code generation time.
```

Listing 7-13 shows FOR statement syntax error handling.

FOR statement syntax error handling

```
001 BEGIN {FOR syntax errors}
002     FOR i := 1, 10 DO five := 5;
                     ^
*** Unexpected token [at ","]
                 ^
*** Missing TO or DOWNTO [at "10"]
003     FOR i = 10 DOWNTO 1 five = 5
              ^
*** Unexpected token [at "="]
              ^
*** Missing := [at "10"]
                          ^
*** Missing DO [at "five"]
                               ^
*** Unexpected token [at "="]
                               ^
*** Missing := [at "5"]
004 END.

          4 source lines.
          7 syntax errors.
          0.05 seconds total parsing time.
```

Design Note

The LOOP node allows the intermediate code to accommodate different flavors of looping constructs in a language-independent way.
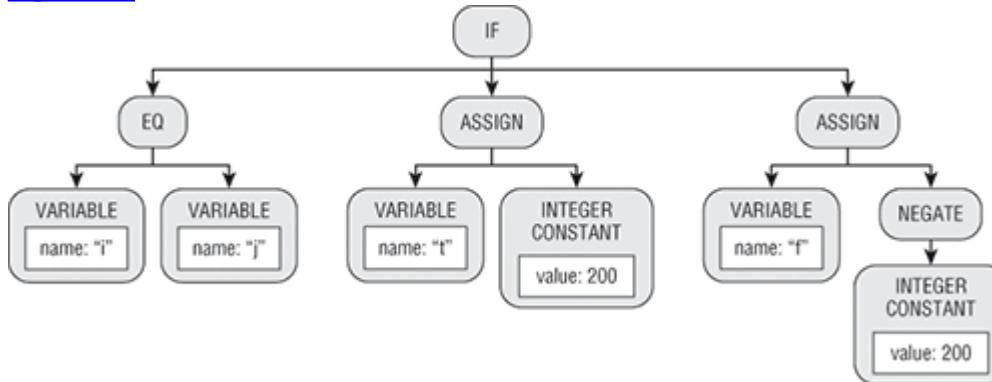
Parsing the IF Statement

The statement parser subclass IfStatementParser parses the Pascal IF statement and generates its parse tree. For the statement

```
IF (i = j) THEN t := 200
           ELSE f := -200;
```

the parse() method generates the parse tree shown in Figure 7-6.

Figure 7-6: Parse tree for an IF statement

The IF node has either two or three children. The first child is the relational expression subtree and the second child is the subtree of the THEN nested statement. If there is an ELSE part, the third child is the subtree of the ELSE nested statement. Otherwise, the IF node has only two children.

Listing 7-14 shows the parse() method of class IfStatementParser.

Listing 7-14: **Method** `parse()` **of class** `IfStatementParser`

```
       // Synchronization set for THEN.
       private static final EnumSet<PascalTokenType> THEN_SET =
           StatementParser.STMT_START_SET.clone();
       static {
           THEN_SET.add(THEN);
           THEN_SET.addAll(StatementParser.STMT_FOLLOW_SET);
       }

       /**
        * Parse an IF statement.
        * @param token the initial token.
        * @return the root node of the generated parse tree.
        * @throws Exception if an error occurred.
        */
     public ICodeNode parse(Token token)
         throws Exception
     {
           token = nextToken();  // consume the IF

           // Create an IF node.
           ICodeNode ifNode = ICodeFactory.createICodeNode(ICodeNodeTypeImpl.IF);

           // Parse the expression.
           // The IF node adopts the expression subtree as its first child.
           ExpressionParser expressionParser = new ExpressionParser(this);
           ifNode.addChild(expressionParser.parse(token));

           // Synchronize at the THEN.
           token = synchronize(THEN_SET);
           if (token.getType() == THEN) {
               token = nextToken();  // consume the THEN
           }
           else {
               errorHandler.flag(token, MISSING_THEN, this);
           }

           // Parse the THEN statement.
           // The IF node adopts the statement subtree as its second child.
           StatementParser statementParser = new StatementParser(this);
           ifNode.addChild(statementParser.parse(token));
           token = currentToken();

           // Look for an ELSE.
           if (token.getType() == ELSE) {
               token = nextToken();  // consume the THEN

               // Parse the ELSE statement.
               // The IF node adopts the statement subtree as its third child.
               ifNode.addChild(statementParser.parse(token));
           }

           return ifNode;
     }
```

The parse() method creates an IF node and builds the parse tree shown in Figure 7-6. It uses the synchronization set THEN_SET to synchronize itself at the THEN.

Listing 7-15 shows syntax checker output from compiling IF statements.

Listing 7-15: **Output from "compiling"** IF **statements**

```
001 BEGIN {IF statements}
002     i := 3;  j := 4;
003
004     IF i = j THEN t := 200
005                ELSE f := -200;
006
007     IF i < j THEN t := 300;
008
009     {Cascading IF THEN ELSEs.}
010     IF      i = 1 THEN f := 10
011     ELSE IF i = 2 THEN f := 20
012     ELSE IF i = 3 THEN t := 30
013     ELSE IF i = 4 THEN f := 40
014     ELSE              f := -1;
015
016     {The "dandling ELSE".}
017     IF i = 3 THEN IF j = 2 THEN t := 500 ELSE f := -500;
018 END.

              18 source lines.
               0 syntax errors.
              0.05 seconds total parsing time.

===== INTERMEDIATE CODE =====

<COMPOUND line="1">
    <ASSIGN line="2">
        <VARIABLE id="i" level="0" />
        <INTEGER_CONSTANT value="3" />
    </ASSIGN>
    <ASSIGN line="2">
        <VARIABLE id="j" level="0" />
        <INTEGER_CONSTANT value="4" />
    </ASSIGN>
    <IF line="4">
        <EQ>
            <VARIABLE id="i" level="0" />
            <VARIABLE id="j" level="0" />
        </EQ>
        <ASSIGN line="4">
            <VARIABLE id="t" level="0" />
            <INTEGER_CONSTANT value="200" />
        </ASSIGN>
        <ASSIGN line="5">
            <VARIABLE id="f" level="0" />
            <NEGATE>
                <INTEGER_CONSTANT value="200" />
            </NEGATE>
        </ASSIGN>
    </IF>
```

```
        <IF line="7">
            <LT>
                    <VARIABLE id="i" level="0" />
                    <VARIABLE id="j" level="0" />
            </LT>
            <ASSIGN line="7">
                    <VARIABLE id="t" level="0" />
                    <INTEGER_CONSTANT value="300" />
            </ASSIGN>
        </IF>
        <IF line="10">
            <EQ>
                    <VARIABLE id="i" level="0" />
                    <INTEGER_CONSTANT value="1" />
            </EQ>
            <ASSIGN line="10">
                    <VARIABLE id="f" level="0" />
                    <INTEGER_CONSTANT value="10" />
            </ASSIGN>
            <IF line="11">
                <EQ>
                    <VARIABLE id="i" level="0" />
                    <INTEGER_CONSTANT value="2" />
                </EQ>
                <ASSIGN line="11">
                    <VARIABLE id="f" level="0" />
                    <INTEGER_CONSTANT value="20" />
                </ASSIGN>
                <IF line="12">
                    <EQ>
                        <VARIABLE id="i" level="0" />
                        <INTEGER_CONSTANT value="3" />
                    </EQ>
                    <ASSIGN line="12">
                        <VARIABLE id="t" level="0" />
                        <INTEGER_CONSTANT value="30" />
                    </ASSIGN>
                    <IF line="13">
                        <EQ>
                            <VARIABLE id="i" level="0" />
                            <INTEGER_CONSTANT value="4" />
                        </EQ>
                        <ASSIGN line="13">
                            <VARIABLE id="f" level="0" />
                            <INTEGER_CONSTANT value="40" />
                        </ASSIGN>
                        <ASSIGN line="14">
                            <VARIABLE id="f" level="0" />
                            <NEGATE>
                                <INTEGER_CONSTANT value="1" />
                            </NEGATE>
                        </ASSIGN>
                    </IF>
                </IF>
            </IF>
        </IF>
        <IF line="17">
            <EQ>
                    <VARIABLE id="i" level="0" />
                    <INTEGER_CONSTANT value="3" />
            </EQ>
            <IF line="17">
                <EQ>
                    <VARIABLE id="j" level="0" />
                    <INTEGER_CONSTANT value="2" />
                </EQ>
                <ASSIGN line="17">
                    <VARIABLE id="t" level="0" />
                    <INTEGER_CONSTANT value="500" />
                </ASSIGN>
                <ASSIGN line="17">
                    <VARIABLE id="f" level="0" />
                    <NEGATE>
                        <INTEGER_CONSTANT value="500" />
                    </NEGATE>
                </ASSIGN>
            </IF>
        </IF>
    </COMPOUND>

                0 instructions generated.
                0.00 seconds total code generation time.
```

The parse tree in Listing 7-15 shows how to parse cascading IF THEN ELSE statements. In each ELSE IF branch, the nested IF statement is treated as any other statement.

The "dangling ELSE" is more problematical. In source line 017, the ELSE can potentially pair with either the first or the second THEN. According to the syntax diagram in Figure 7-1, the IF statement

```
IF (j = 2) THEN t := 500 ELSE f := -500
```

is the nested THEN statement of

```
IF (i = 1) THEN
```

Therefore, the ELSE pairs with the second THEN.

Listing 7-16 shows IF statement syntax error handling.

Listing 7-16: IF statement syntax error handling

```
001 BEGIN {IF syntax errors}
002     i := 0;
003
004     IF i = 5;
                   ^
*** Missing THEN [at ";"]
005     IF i := 5 ELSE j := 9;
               ^
```

```
*** Unexpected token [at ":="]
                  ^
*** Missing THEN [at "ELSE"]
006     IF i = 5 ELSE j := 9 THEN j := 7;
                  ^
*** Missing THEN [at "ELSE"]
                         ^
*** Unexpected token [at "THEN"]
007 END.

            7 source lines.
            5 syntax errors.
        0.05 seconds total parsing time.
```

Parsing the CASE Statement

The CASE statement is the most challenging of the Pascal control statements to parse. The statement parser subclass CompoundStatementParser parses the CASE statement and generates its parse tree. For the statement
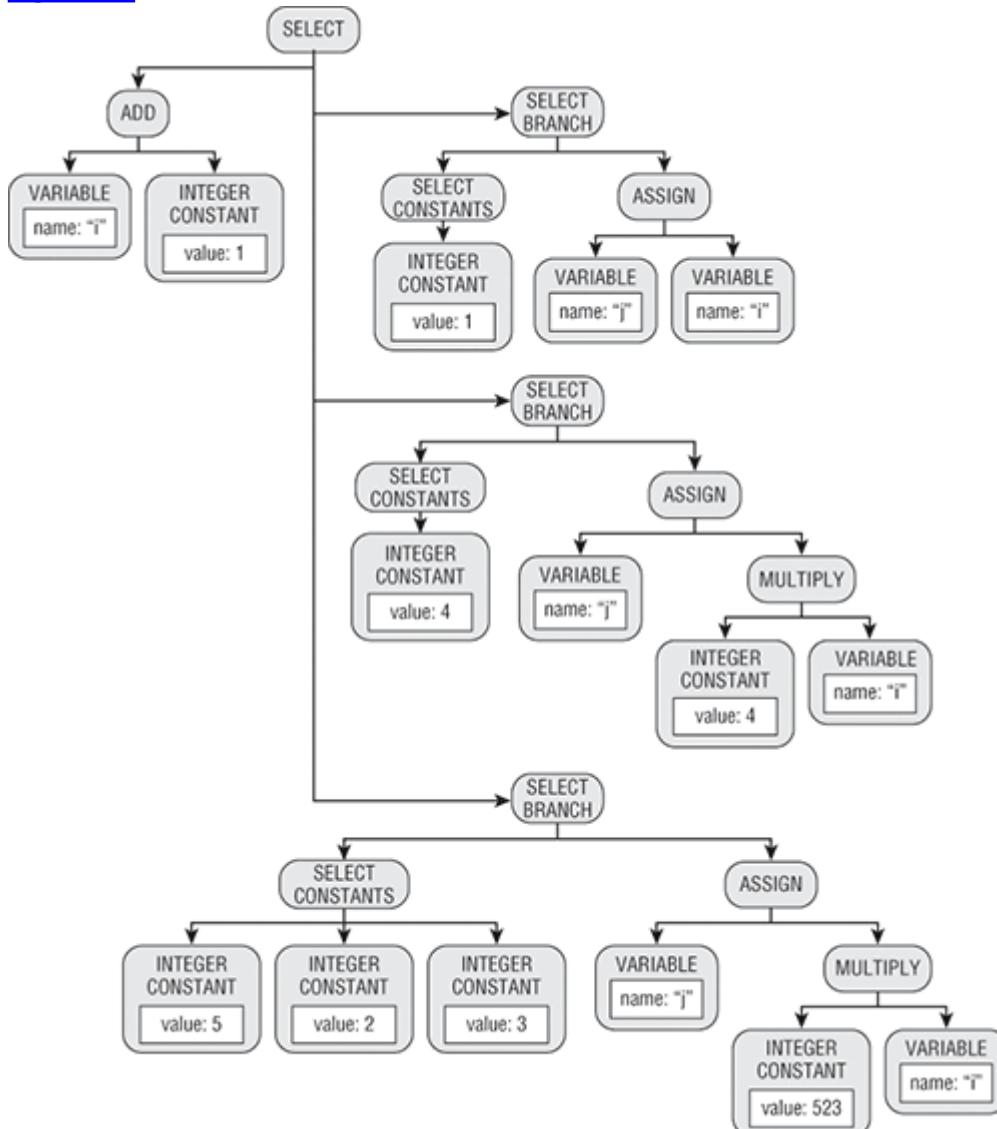
```
CASE i+1 OF
    1:      j := i;
    4:      j := 4*i;
    5, 2, 3: j := 523*i;
END
```

the parse() method generates the parse tree shown in Figure 7-7.

Listing 7-17 shows the parse() method of class CaseStatementParser.

Figure 7-7: Parse tree for a CASE statement

Listing 7-17: Method **parse()** of class **CaseStatementParser**

```
// Synchronization set for starting a CASE option constant.
private static final EnumSet<PascalTokenType> CONSTANT_START_SET =
    EnumSet.of(IDENTIFIER, INTEGER, PLUS, MINUS, STRING);

// Synchronization set for OF.
private static final EnumSet<PascalTokenType> OF_SET =
    CONSTANT_START_SET.clone();
static {
    OF_SET.add(OF);
    OF_SET.addAll(StatementParser.STMT_FOLLOW_SET);
}

/**
 * Parse a CASE statement.
 * @param token the initial token.
```

```
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token)
    throws Exception
{
    token = nextToken();  // consume the CASE

    // Create a SELECT node.
    ICodeNode selectNode = ICodeFactory.createICodeNode(SELECT);

    // Parse the CASE expression.
    // The SELECT node adopts the expression subtree as its first child.
    ExpressionParser expressionParser = new ExpressionParser(this);
    selectNode.addChild(expressionParser.parse(token));

    // Synchronize at the OF.
    token = synchronize(OF_SET);
    if (token.getType() == OF) {
        token = nextToken();  // consume the OF
    }
    else {
        errorHandler.flag(token, MISSING_OF, this);
    }

    // Set of CASE branch constants.
    HashSet<Object> constantSet = new HashSet<Object>();

    // Loop to parse each CASE branch until the END token
    // or the end of the source file.
    while (!(token instanceof EofToken) && (token.getType() != END)) {

        // The SELECT node adopts the CASE branch subtree.
        selectNode.addChild(parseBranch(token, constantSet));

        token = currentToken();
        TokenType tokenType = token.getType();

        // Look for the semicolon between CASE branches.
        if (tokenType == SEMICOLON) {
            token = nextToken();  // consume the ;
        }

        // If at the start of the next constant, then missing a semicolon.
        else if (CONSTANT_START_SET.contains(tokenType)) {
            errorHandler.flag(token, MISSING_SEMICOLON, this);
        }
    }

    // Look for the END token.
    if (token.getType() == END) {
        token = nextToken();  // consume END
    }
    else {
        errorHandler.flag(token, MISSING_END, this);
    }

    return selectNode;
}
```

The synchronization set CONSTANT_START_SET contains the token types that can start a constant. Method parse() creates a SELECT node, and after parsing the CASE expression, it uses the synchronization set OF_SET to synchronize itself at the OF token. The while loop calls parseBranch() to parse each CASE branch and then looks for a semicolon. The loop exits with the END token. The method returns the SELECT node.

Listing 7-18 shows method parseBranch().

Listing 7-18: Method parseBranch() of class CaseStatementParser

```
/**
 * Parse a CASE branch.
 * @param token the current token.
 * @param constantSet the set of CASE branch constants.
 * @return the root SELECT_BRANCH node of the subtree.
 * @throws Exception if an error occurred.
 */
private ICodeNode parseBranch(Token token, HashSet<Object> constantSet)
    throws Exception
{
    // Create an SELECT_BRANCH node and a SELECT_CONSTANTS node.
    // The SELECT_BRANCH node adopts the SELECT_CONSTANTS node as its
    // first child.
    ICodeNode branchNode = ICodeFactory.createICodeNode(SELECT_BRANCH);
    ICodeNode constantsNode =
                    ICodeFactory.createICodeNode(SELECT_CONSTANTS);
    branchNode.addChild(constantsNode);

    // Parse the list of CASE branch constants.
    // The SELECT_CONSTANTS node adopts each constant.
    parseConstantList(token, constantsNode, constantSet);

    // Look for the : token.
    token = currentToken();
    if (token.getType() == COLON) {
        token = nextToken();  // consume the :
    }
    else {
        errorHandler.flag(token, MISSING_COLON, this);
    }

    // Parse the CASE branch statement. The SELECT_BRANCH node adopts
    // the statement subtree as its second child.
    StatementParser statementParser = new StatementParser(this);
    branchNode.addChild(statementParser.parse(token));

    return branchNode;
}
```

Method parseBranch() creates a SELECT_BRANCH node and a SELECT_CONSTANTS node before calling parseConstantList() to parse the list of constants. It then looks for a colon. The SELECT_CONSTANTS node adopts the constant nodes and the SELECT_BRANCH node adopts the SELECT_CONSTANTS node. A call to statementParser.parse() parses the branch statement, and the SELECT_BRANCH node adopts the root of the branch statement's parse tree as its second child.

Listing 7-19 shows method parseConstantList().

```
    // Synchronization set for COMMA.
    private static final EnumSet<PascalTokenType> COMMA_SET =
        CONSTANT_START_SET.clone();
    static {
        COMMA_SET.add(COMMA);
        COMMA_SET.add(COLON);
        COMMA_SET.addAll(StatementParser.STMT_START_SET);
        COMMA_SET.addAll(StatementParser.STMT_FOLLOW_SET);
    }

    /**
     * Parse a list of CASE branch constants.
     * @param token the current token.
     * @param constantsNode the parent SELECT_CONSTANTS node.
     * @param constantSet the set of CASE branch constants.
     * @throws Exception if an error occurred.
     */
    private void parseConstantList(Token token, ICodeNode constantsNode,
                                   HashSet<Object> constantSet)
        throws Exception
    {
        // Loop to parse each constant.
        while (CONSTANT_START_SET.contains(token.getType())) {

            // The constants list node adopts the constant node.
            constantsNode.addChild(parseConstant(token, constantSet));

            // Synchronize at the comma between constants.
            token = synchronize(COMMA_SET);

            // Look for the comma.
            if (token.getType() == COMMA) {
                token = nextToken();  // consume the ,
            }

            // If at the start of the next constant, then missing a comma.
            else if (CONSTANT_START_SET.contains(token.getType())) {
                errorHandler.flag(token, MISSING_COMMA, this);
            }
        }
    }
```

Method `parseConstantList()` parses a list of constants separated by commas. It calls `parseConstant()` to parse each constant. In this chapter, `parseConstant()` can parse integer constants (optionally preceded by a + or – sign) by calling `parseIntegerConstant()` and character constants by calling `parseCharacterConstant()`. Since you haven't yet parsed constant declarations, named constants (`IDENTIFIER` tokens) are not allowed until a later chapter.

Listing 7-20 shows the constant parsing methods.

Listing 7-20: Constant parsing methods of class `CaseStatementParser`

```
    /**
     * Parse CASE branch constant.
     * @param token the current token.
     * @param constantSet the set of CASE branch constants.
     * @return the constant node.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseConstant(Token token, HashSet<Object> constantSet)
        throws Exception
    {
        TokenType sign = null;
        ICodeNode constantNode = null;

        // Synchronize at the start of a constant.
        token = synchronize(CONSTANT_START_SET);
        TokenType tokenType = token.getType();

        // Plus or minus sign?
        if ((tokenType == PLUS) || (tokenType == MINUS)) {
            sign = tokenType;
            token = nextToken();  // consume sign
        }

        // Parse the constant.
        switch ((PascalTokenType) token.getType()) {

            case IDENTIFIER: {
                constantNode = parseIdentifierConstant(token, sign);
                break;
            }

            case INTEGER: {
                constantNode = parseIntegerConstant(token.getText(), sign);
                break;
            }

            case STRING: {
                constantNode =
                    parseCharacterConstant(token, (String) token.getValue(),
                                           sign);
                break;
            }

            default: {
                errorHandler.flag(token, INVALID_CONSTANT, this);
                break;
            }
        }

        // Check for reused constants.
        if (constantNode != null) {
            Object value = constantNode.getAttribute(VALUE);

            if (constantSet.contains(value)) {
                errorHandler.flag(token, CASE_CONSTANT_REUSED, this);
            }
            else {
                constantSet.add(value);
            }
        }

        nextToken();  // consume the constant
        return constantNode;
    }
```

```java
        /**
         * Parse an identifier CASE constant.
         * @param value the current token value string.
         * @param sign the sign, if any.
         * @return the constant node.
         */
        private ICodeNode parseIdentifierConstant(Token token, TokenType sign)
            throws Exception
        {
            // Placeholder: Don&apos;t allow for now.
            errorHandler.flag(token, INVALID_CONSTANT, this);
            return null;
        }

        /**
         * Parse an integer CASE constant.
         * @param value the current token value string.
         * @param sign the sign, if any.
         * @return the constant node.
         */
        private ICodeNode parseIntegerConstant(String value, TokenType sign)
        {
            ICodeNode constantNode = ICodeFactory.createICodeNode(INTEGER_CONSTANT);
            int intValue = Integer.parseInt(value);

            if (sign == MINUS) {
                intValue = -intValue;
            }

            constantNode.setAttribute(VALUE, intValue);
            return constantNode;
        }

        /**
         * Parse a character CASE constant.
         * @param token the current token.
         * @param value the token value string.
         * @param sign the sign, if any.
         * @return the constant node.
         */
        private ICodeNode parseCharacterConstant(Token token, String value,
                                                 TokenType sign)
        {
            ICodeNode constantNode = null;

            if (sign != null) {
                errorHandler.flag(token, INVALID_CONSTANT, this);
            }
            else {
                if (value.length() == 1) {
                    constantNode = ICodeFactory.createICodeNode(STRING_CONSTANT);
                    constantNode.setAttribute(VALUE, value);
                }
                else {
                    errorHandler.flag(token, INVALID_CONSTANT, this);
                }
            }

            return constantNode;
        }
```

The syntax diagram for the CASE statement in Figure 7-1 does not reflect two syntax rules regarding a branch constant:

- A branch constant can be an integer or a single character.
- A branch constant cannot be used more than once in a CASE statement.

To address the first rule, method parseConstant() checks that any STRING token has a value of length 1. For the second rule, method parse() creates the set constantSet, which is passed down to method parseConstant(). When it is done parsing the constant, method parseConstant() checks the set to ensure that the constant has not already been used.

Listing 7-21 shows syntax checker output from compiling CASE statements.

Listing 7-21: Output from "compiling" CASE statements

```
001 BEGIN {CASE statements}
002     i := 3;  ch := &apos;b&apos;;
003
004     CASE i+1 OF
005         1:        j := i;
006         4:        j := 4*i;
007         5, 2, 3: j := 523*i;
008     END;
009
010     CASE ch OF
011         &apos;c&apos;, &apos;b&apos; : str := &apos;p&apos;;
012         &apos;a&apos;       : str := &apos;q&apos;
013     END;
014
015     FOR i := -5 TO 15 DO BEGIN
016         CASE i OF
017             2: prime := i;
018             -4, -2, 0, 4, 6, 8, 10, 12: even := i;
019             -3, -1, 1, 3, 5, 7, 9, 11: CASE i OF
020                                             -3, -1, 1, 9:   odd := i;
021                                             2, 3, 5, 7, 11: prime := i;
022                                        END
023         END
024     END
025 END.

                  25 source lines.
                   0 syntax errors.
                  0.06 seconds total parsing time.

===== INTERMEDIATE CODE =====

<COMPOUND line="1">
    <ASSIGN line="2">
        <VARIABLE id="i" level="0" />
        <INTEGER_CONSTANT value="3" />
    </ASSIGN>
    <ASSIGN line="2">
        <VARIABLE id="ch" level="0" />
        <STRING_CONSTANT value="b" />
    </ASSIGN>
    <SELECT line="4">
        <ADD>
```

```xml
                    <VARIABLE id="i" level="0" />
                    <INTEGER_CONSTANT value="1" />
                </ADD>
            <SELECT_BRANCH>
                <SELECT_CONSTANTS>
                    <INTEGER_CONSTANT value="1" />
                </SELECT_CONSTANTS>
                <ASSIGN line="5">
                    <VARIABLE id="j" level="0" />
                    <VARIABLE id="i" level="0" />
                </ASSIGN>
            </SELECT_BRANCH>
            <SELECT_BRANCH>
                <SELECT_CONSTANTS>
                    <INTEGER_CONSTANT value="4" />
                </SELECT_CONSTANTS>
                <ASSIGN line="6">
                    <VARIABLE id="j" level="0" />
                    <MULTIPLY>
                        <INTEGER_CONSTANT value="4" />
                        <VARIABLE id="i" level="0" />
                    </MULTIPLY>
                </ASSIGN>
            </SELECT_BRANCH>
            <SELECT_BRANCH>
                <SELECT_CONSTANTS>
                    <INTEGER_CONSTANT value="5" />
                    <INTEGER_CONSTANT value="2" />
                    <INTEGER_CONSTANT value="3" />
                </SELECT_CONSTANTS>
                <ASSIGN line="7">
                    <VARIABLE id="j" level="0" />
                    <MULTIPLY>
                        <INTEGER_CONSTANT value="523" />
                        <VARIABLE id="i" level="0" />
                    </MULTIPLY>
                </ASSIGN>
            </SELECT_BRANCH>
        </SELECT>
        <SELECT line="10">
            <VARIABLE id="ch" level="0" />
            <SELECT_BRANCH>
                <SELECT_CONSTANTS>
                    <STRING_CONSTANT value="c" />
                    <STRING_CONSTANT value="b" />
                </SELECT_CONSTANTS>
                <ASSIGN line="11">
                    <VARIABLE id="str" level="0" />
                    <STRING_CONSTANT value="p" />
                </ASSIGN>
            </SELECT_BRANCH>
            <SELECT_BRANCH>
                <SELECT_CONSTANTS>
                    <STRING_CONSTANT value="a" />
                </SELECT_CONSTANTS>
                <ASSIGN line="12">
                    <VARIABLE id="str" level="0" />
                    <STRING_CONSTANT value="q" />
                </ASSIGN>
            </SELECT_BRANCH>
        </SELECT>
        <COMPOUND line="15">
            <ASSIGN line="15">
                <VARIABLE id="i" level="0" />
                <NEGATE>
                    <INTEGER_CONSTANT value="5" />
                </NEGATE>
            </ASSIGN>
            <LOOP>
                <TEST>
                    <GT>
                        <VARIABLE id="i" level="0" />
                        <INTEGER_CONSTANT value="15" />
                    </GT>
                </TEST>
                <COMPOUND line="15">
                    <SELECT line="16">
                        <VARIABLE id="i" level="0" />
                        <SELECT_BRANCH>
                            <SELECT_CONSTANTS>
                                <INTEGER_CONSTANT value="2" />
                            </SELECT_CONSTANTS>
                            <ASSIGN line="17">
                                <VARIABLE id="prime" level="0" />
                                <VARIABLE id="i" level="0" />
                            </ASSIGN>
                        </SELECT_BRANCH>
                        <SELECT_BRANCH>
                            <SELECT_CONSTANTS>
                                <INTEGER_CONSTANT value="-4" />
                                <INTEGER_CONSTANT value="-2" />
                                <INTEGER_CONSTANT value="0" />
                                <INTEGER_CONSTANT value="4" />
                                <INTEGER_CONSTANT value="6" />
                                <INTEGER_CONSTANT value="8" />
                                <INTEGER_CONSTANT value="10" />
                                <INTEGER_CONSTANT value="12" />
                            </SELECT_CONSTANTS>
                            <ASSIGN line="18">
                                <VARIABLE id="even" level="0" />
                                <VARIABLE id="i" level="0" />
                            </ASSIGN>
                        </SELECT_BRANCH>
                        <SELECT_BRANCH>
                            <SELECT_CONSTANTS>
                                <INTEGER_CONSTANT value="-3" />
                                <INTEGER_CONSTANT value="-1" />
                                <INTEGER_CONSTANT value="1" />
                                <INTEGER_CONSTANT value="3" />
                                <INTEGER_CONSTANT value="5" />
                                <INTEGER_CONSTANT value="7" />
                                <INTEGER_CONSTANT value="9" />
                                <INTEGER_CONSTANT value="11" />
                            </SELECT_CONSTANTS>
                            <SELECT line="19">
                                <VARIABLE id="i" level="0" />
                                <SELECT_BRANCH>
                                    <SELECT_CONSTANTS>
                                        <INTEGER_CONSTANT value="-3" />
                                        <INTEGER_CONSTANT value="-1" />
                                        <INTEGER_CONSTANT value="1" />
                                        <INTEGER_CONSTANT value="9" />
```

```
                                    </SELECT_CONSTANTS>
                                    <ASSIGN line="20">
                                        <VARIABLE id="odd" level="0" />
                                        <VARIABLE id="i" level="0" />
                                    </ASSIGN>
                                </SELECT_BRANCH>
                                <SELECT_BRANCH>
                                    <SELECT_CONSTANTS>
                                        <INTEGER_CONSTANT value="2" />
                                        <INTEGER_CONSTANT value="3" />
                                        <INTEGER_CONSTANT value="5" />
                                        <INTEGER_CONSTANT value="7" />
                                        <INTEGER_CONSTANT value="11" />
                                    </SELECT_CONSTANTS>
                                    <ASSIGN line="21">
                                        <VARIABLE id="prime" level="0" />
                                        <VARIABLE id="i" level="0" />
                                    </ASSIGN>
                                </SELECT_BRANCH>
                            </SELECT>
                        </SELECT_BRANCH>
                    </SELECT>
            </COMPOUND>
            <ASSIGN line="15">
                <VARIABLE id="i" level="0" />
                <ADD>
                    <VARIABLE id="i" level="0" />
                    <INTEGER_CONSTANT value="1" />
                </ADD>
            </ASSIGN>
        </LOOP>
    </COMPOUND>
</COMPOUND>

                0 instructions generated.
                0.00 seconds total code generation time.
```

[Listing 7-22](#) shows CASE statement syntax error handling.

Listing 7-22: CASE statement syntax error handling

```
001 BEGIN {CASE syntax errors}
002     i := 0;  ch := 'x';  str := 'y';
003
004     CASE i OF
005         1 2 3: j := i;
                   ^
*** Missing , [at "2"]
                     ^
*** Missing , [at "3"]
006         4,1,5  IF j = 5 THEN k := 9;
                   ^
*** CASE constant reused [at "1"]
                          ^
*** Missing : [at "IF"]
007     END;
008
009     CASE ch1 OF
                ^
*** Undefined identifier [at "ch1"]
010         'x', 'hello', 'y': str := 'world';
                  ^
*** Invalid constant [at "'hello'"]
011         'z', 'x':           str := 'bye';
                  ^
*** CASE constant reused [at "'x'"]
012     END
013 END.

                13 source lines.
                 7 syntax errors.
                0.05 seconds total parsing time.
```

In the next chapter, you'll develop executors in the interpreter back end to execute these control statements.

Chapter 8

Interpreting Control Statements

In this chapter, you'll extend your work from Chapter 6 by developing executors in the interpreter back end for the Pascal control statements parsed in Chapter 7.

Goals and Approach

This chapter has one major goal:

- Language-independent executors in the interpreter back end that will interpret the intermediate code and execute control statements.

The approach in this chapter will be to develop more subclasses of class `Executor`. To verify our work, you'll continue to use the simple interpreter program from Chapter 6.

Program 8: Simple Interpreter II
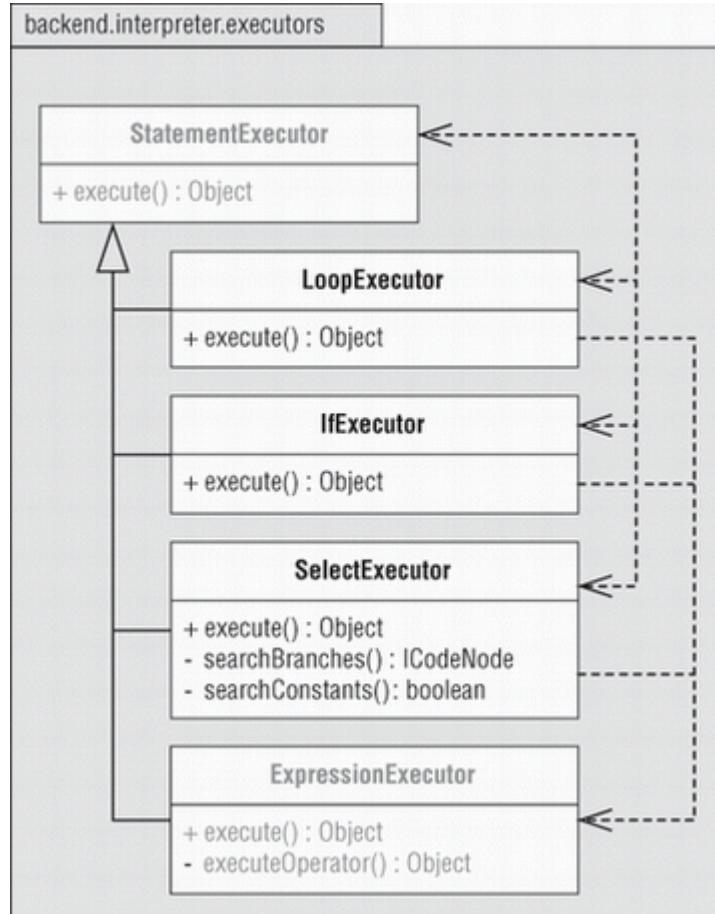
The main class Pascal will continue to be unchanged from Chapters 6 and 7. In this chapter, you'll go back to running it as an interpreter. The interpreting capabilities of the program as a whole will increase with the addition of the new executor subclasses for the control statements. You'll run the program several times this chapter on source files containing various Pascal control statements to verify that the new subclasses can properly execute those statements.

Interpreting Control Statements

The UML diagram in
Figure 8-1 adds to the diagram in Figure 6-1. Compare this figure with Figure 7-2.

Figure 8-1: The control statement executor subclasses

The executor subclass `StatementExecutor` depends on its subclasses, and they each in turn depend on `StatementExecutor`. Subclasses `LoopExecutor`, `IfExecutor`, and `SelectExecutor` also depend on `ExpressionExecutor`.

Listing 8-1 shows an updated version of method `execute()` of `StatementExecutor` which now can handle LOOP, IF, and SELECT nodes.

Listing 8-1: Method `execute()` of class `StatementExecutor`

```java
/**
 * Execute a statement.
 * To be overridden by the specialized statement executor subclasses.
 * @param node the root node of the statement.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    ICodeNodeTypeImpl nodeType = (ICodeNodeTypeImpl) node.getType();

    // Send a message about the current source line.
    sendMessage(node);

    switch (nodeType) {

        case COMPOUND: {
            CompoundExecutor compoundExecutor = new CompoundExecutor(this);
            return compoundExecutor.execute(node);
        }

        case ASSIGN: {
            AssignmentExecutor assignmentExecutor =
                new AssignmentExecutor(this);
            return assignmentExecutor.execute(node);
        }

        case LOOP: {
            LoopExecutor loopExecutor = new LoopExecutor(this);
            return loopExecutor.execute(node);
        }

        case IF: {
            IfExecutor ifExecutor = new IfExecutor(this);
            return ifExecutor.execute(node);
        }

        case SELECT: {
            SelectExecutor selectExecutor = new SelectExecutor(this);
            return selectExecutor.execute(node);
        }

        case NO_OP: return null;

        default: {
            errorHandler.flag(node, UNIMPLEMENTED_FEATURE, this);
            return null;
        }
    }
}
```

Executing a Looping Statement

Listing 8-2 shows the `execute()` method of the statement executor subclass `LoopExecutor`, which executes LOOP parse trees.

Listing 8-2: Method `execute()` of class `LoopExecutor`

```java
/**
 * Execute a loop statement.
 * @param node the root node of the statement.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    boolean exitLoop = false;
    ICodeNode exprNode = null;
    List<ICodeNode> loopChildren = node.getChildren();

    ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
    StatementExecutor statementExecutor = new StatementExecutor(this);

    // Loop until the TEST expression value is true.
    while (!exitLoop) {
        ++executionCount;  // count the loop statement itself

        // Execute the children of the LOOP node.
        for (ICodeNode child : loopChildren) {
            ICodeNodeTypeImpl childType =
                                (ICodeNodeTypeImpl) child.getType();

            // TEST node?
            if (childType == TEST) {
                if (exprNode == null) {
                    exprNode = child.getChildren().get(0);
                }
                exitLoop = (Boolean) expressionExecutor.execute(exprNode);
            }

            // Statement node.
            else {
                statementExecutor.execute(child);
            }

            // Exit if the TEST expression value is true,
            if (exitLoop) {
                break;
            }
        }
    }

    return null;
}
```

The `execute()` method's `while` loop repeatedly interprets the children of the `LOOP` node. The inner `for` loop executes each child that is a statement subtree. One of the children can be a `TEST` node whose child is a relative expression subtree. The method executes the expression and exits the loop if the expression evaluates to true.

[Listing 8-3](#) shows the output from executing a Pascal `REPEAT` statement using a command line similar to

```
java -classpath classes Pascal execute repeat.txt
```

[Listing 8-3:](#) Executing a Pascal `REPEAT` statement

```
001 BEGIN {Calculate the square root of 4 using Newton&apos;s method.}
002     number := 4;
003     root := number;
004
005     REPEAT
006         partial := number/root + root;
007         root := partial/2
008     UNTIL root*root - number < 0.000001
009 END.

                   9 source lines.
                   0 syntax errors.
                0.05 seconds total parsing time.


===== OUTPUT =====

>>> LINE 002: number = 4
>>> LINE 003: root = 4
>>> LINE 006: partial = 5.0
>>> LINE 007: root = 2.5
>>> LINE 006: partial = 4.1
>>> LINE 007: root = 2.05
>>> LINE 006: partial = 4.0012197
>>> LINE 007: root = 2.0006099
>>> LINE 006: partial = 4.0
>>> LINE 007: root = 2.0

                  14 statements executed.
                   0 runtime errors.
                0.01 seconds total execution time.
```

[Listing 8-4](#) shows the output from executing a Pascal `WHILE` statement.

[Listing 8-4:](#) Executing a Pascal `WHILE` statement

```
001 BEGIN {Calculate the square root of 2 using Newton&apos;s method.}
002     number := 2;
003     root := number;
004
005     WHILE root*root - number > 0.000001 DO BEGIN
006         root := (number/root + root)/2
007     END
008 END.

                   8 source lines.
                   0 syntax errors.
                0.05 seconds total parsing time.


===== OUTPUT =====

>>> LINE 002: number = 2
>>> LINE 003: root = 2
>>> LINE 006: root = 1.5
>>> LINE 006: root = 1.4166667
>>> LINE 006: root = 1.4142157
>>> LINE 006: root = 1.4142135

                  11 statements executed.
                   0 runtime errors.
                0.01 seconds total execution time.
```

[Listing 8-5](#) shows the output from executing some Pascal `FOR` statements.

[Listing 8-5:](#) Executing Pascal `FOR` statements

```
001 BEGIN {FOR statements}
002     j := 1;
003
004     FOR k := j TO 5 DO n := k;
005
006     FOR k := n DOWNTO 1 DO j := k;
007
008     FOR i := 1 TO 2 DO BEGIN
009         FOR j := 1 TO 3 DO BEGIN
010             k := i*j
011         END
012     END
013 END.

                  13 source lines.
                   0 syntax errors.
                0.05 seconds total parsing time.

===== OUTPUT =====

>>> LINE 002: j = 1
>>> LINE 004: k = 1
>>> LINE 004: n = 1
>>> LINE 004: k = 2
>>> LINE 004: n = 2
>>> LINE 004: k = 3
>>> LINE 004: n = 3
>>> LINE 004: k = 4
>>> LINE 004: n = 4
>>> LINE 004: k = 5
>>> LINE 004: n = 5
```

```
>>> LINE 004: k = 6
>>> LINE 006: k = 5
>>> LINE 006: j = 5
>>> LINE 006: k = 4
>>> LINE 006: j = 4
>>> LINE 006: k = 3
>>> LINE 006: j = 3
>>> LINE 006: k = 2
>>> LINE 006: j = 2
>>> LINE 006: k = 1
>>> LINE 006: j = 1
>>> LINE 006: k = 0
>>> LINE 008: i = 1
>>> LINE 009: j = 1
>>> LINE 010: k = 1
>>> LINE 009: j = 2
>>> LINE 010: k = 2
>>> LINE 009: j = 3
>>> LINE 010: k = 3
>>> LINE 009: j = 4
>>> LINE 008: i = 2
>>> LINE 009: j = 1
>>> LINE 010: k = 2
>>> LINE 009: j = 2
>>> LINE 010: k = 4
>>> LINE 009: j = 3
>>> LINE 010: k = 6
>>> LINE 009: j = 4
>>> LINE 008: i = 3

           63 statements executed.
            0 runtime errors.
         0.03 seconds total execution time.
```

Design Note

As noted in the previous chapter, the intermediate code accommodates different flavors of source program looping constructs in a language-independent way with the LOOP node. This enables the executor subclass LoopExecutor to execute Pascal REPEAT, WHILE, and FOR loops.

Executing the IF Statement

Listing 8-6 shows the execute() method of the statement executor subclass IfExecutor, which executes IF parse trees.

Listing 8-6: Method execute() of class IfExecutor

```
/**
 * Execute an IF statement.
 * @param node the root node of the statement.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    // Get the IF node's children.
    List<ICodeNode> children = node.getChildren();
    ICodeNode exprNode = children.get(0);
    ICodeNode thenStmtNode = children.get(1);
    ICodeNode elseStmtNode = children.size() > 2 ? children.get(2) : null;

    ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
    StatementExecutor statementExecutor = new StatementExecutor(this);

    // Evaluate the expression to determine which statement to execute.
    boolean b = (Boolean) expressionExecutor.execute(exprNode);
    if (b) {
        statementExecutor.execute(thenStmtNode);
    }
    else if (elseStmtNode != null) {
        statementExecutor.execute(elseStmtNode);
    }

    ++executionCount;  // count the IF statement itself
    return null;
}
```

The execute() method executes the IF node's first child, which is a relative expression subtree. If the expression evaluates to true, then the method executes the second child, the THEN statement subtree. If the expression evaluates to false *and* there is a third child, the method executes instead the third child, the ELSE statement subtree.

Listing 8-7 shows the output from executing some Pascal IF statements. It properly handles cascading IF THEN ELSE statements and the dangling ELSE.

Listing 8-7: Executing Pascal IF statements

```
001 BEGIN {IF statements}
002     i := 3;  j := 4;
003
004     IF i = j THEN t := 200
005             ELSE f := -200;
006
007     IF i < j THEN t := 300;
008
009     {Cascading IF THEN ELSEs.}
010     IF      i = 1 THEN f := 10
011     ELSE IF i = 2 THEN f := 20
012     ELSE IF i = 3 THEN t := 30
013     ELSE IF i = 4 THEN f := 40
014     ELSE               f := -1;
015
016     {The "dangling ELSE".}
017     IF i = 3 THEN IF j = 2 THEN t := 500 ELSE f := -500;
018 END.
```

```
                18 source lines.
                 0 syntax errors.
             0.05 seconds total parsing time.


===== OUTPUT =====

>>> LINE 002: i = 3
>>> LINE 002: j = 4
>>> LINE 005: f = -200
>>> LINE 007: t = 300
>>> LINE 012: t = 30
>>> LINE 017: f = -500

                13 statements executed.
                 0 runtime errors.
             0.01 seconds total execution time.
```

Executing the SELECT Statement

Listing 8-8 shows the `execute()` method of executor subclass `SelectExecutor`, which executes SELECT parse trees.

Listing 8-8: Method **`execute()`** of class **`SelectExecutor`**

```
/**
 * Execute SELECT statement.
 * @param node the root node of the statement.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    // Get the SELECT node&apos;s children.
    ArrayList<ICodeNode> selectChildren = node.getChildren();
    ICodeNode exprNode = selectChildren.get(0);

    // Evaluate the SELECT expression.
    ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
    Object selectValue = expressionExecutor.execute(exprNode);

    // Attempt to select a SELECT_BRANCH.
    ICodeNode selectedBranchNode = searchBranches(selectValue,
                                                  selectChildren);

    // If there was a selection, execute the SELECT_BRANCH&apos;s statement.
    if (selectedBranchNode != null) {
        ICodeNode stmtNode = selectedBranchNode.getChildren().get(1);
        StatementExecutor statementExecutor = new StatementExecutor(this);
        statementExecutor.execute(stmtNode);
    }

    ++executionCount;  // count the SELECT statement itself
    return null;
}
```

Method `execute()` executes the SELECT node's first child, which is the expression that evaluates to the selection value that `searchBranches()` uses to search the node's SELECT_BRANCH children. If `searchBranches()` returns a reference to a branch, then method `execute()` executes that branch's statement.

Listing 8-9 shows method `searchBranches()`. It attempts to find a branch that has a constant whose value equals the selection value. It loops over the SELECT node's SELECT_BRANCH children. For each branch, it calls `searchConstants()` to search for a matching value among the branch's constants. It returns the SELECT_BRANCH node as soon as there is a match or null if there was no match.

Listing 8-9: Method **`searchBranches()`** of class **`SelectExecutor`**

```
/**
 * Search the SELECT_BRANCHes to find a match.
 * @param selectValue the value to match.
 * @param selectChildren the children of the SELECT node.
 * @return ICodeNode
 */
private ICodeNode searchBranches(Object selectValue,
                                 ArrayList<ICodeNode> selectChildren)
{
    // Loop over the SELECT_BRANCHes to find a match.
    for (int i = 1; i < selectChildren.size(); ++i) {
        ICodeNode branchNode = selectChildren.get(i);

        if (searchConstants(selectValue, branchNode)) {
            return branchNode;
        }
    }

    return null;
}
```

Listing 8-10 shows method `searchConstants()`, which searches the constants that are the SELECT_CONSTANTS node's children for a given SELECT branch. It returns true as soon as it finds a match for the selection value or false if there was no match.

Listing 8-10: Method **`searchConstants()`** of class **`SelectExecutor`**

```
/**
 * Search the constants of a SELECT_BRANCH for a matching value.
 * @param selectValue the value to match.
 * @param branchNode the SELECT_BRANCH node.
 * @return boolean
 */
private boolean searchConstants(Object selectValue, ICodeNode branchNode)
{
    // Are the values integer or string?
    boolean integerMode = selectValue instanceof Integer;

    // Get the list of SELECT_CONSTANTS values.
    ICodeNode constantsNode = branchNode.getChildren().get(0);
    ArrayList<ICodeNode> constantsList = constantsNode.getChildren();

    // Search the list of constants.
    if (selectValue instanceof Integer) {
        for (ICodeNode constantNode : constantsList) {
            int constant = (Integer) constantNode.getAttribute(VALUE);
```

```
                if (((Integer) selectValue) == constant) {
                    return true;  // match
                }
            }
        }
        else {
            for (ICodeNode constantNode : constantsList) {
                String constant = (String) constantNode.getAttribute(VALUE);

                if (((String) selectValue).equals(constant)) {
                    return true;  // match
                }
            }
        }

        return false;  // no match
    }
```

[Listing 8-11](#) shows output from parsing and executing a Pascal CASE statement.

Executing a Pascal CASE statement

```
001 BEGIN {CASE statements}
002    i := 3;   ch := 'b';
003
004    CASE i+1 OF
005        1:      j := i;
006        4:      j := 4*i;
007        5, 2, 3: j := 523*i;
008    END;
009
010    CASE ch OF
011        'c', 'b' : str := 'p';
012        'a'       : str := 'q'
013    END;
014
015    FOR i := -5 TO 15 DO BEGIN
016        CASE i OF
017            2: prime := i;
018            -4, -2, 0, 4, 6, 8, 10, 12: even := i;
019            -3, -1, 1, 3, 5, 7, 9, 11:  CASE i OF
020                                    -3, -1, 1, 9:   odd := i;
021                                    2, 3, 5, 7, 11: prime := i;
022                                END
023        END
024    END
025 END.

                25 source lines.
                 0 syntax errors.
                0.06 seconds total parsing time.

 ===== OUTPUT =====

 >>> LINE 002: i = 3
 >>> LINE 002: ch = b
 >>> LINE 006: j = 12
 >>> LINE 011: str = p
 >>> LINE 015: i = -5
 >>> LINE 015: i = -4
 >>> LINE 018: even = -4
 >>> LINE 015: i = -3
 >>> LINE 020: odd = -3
 >>> LINE 015: i = -2
 >>> LINE 018: even = -2
 >>> LINE 015: i = -1
 >>> LINE 020: odd = -1
 >>> LINE 015: i = 0
 >>> LINE 018: even = 0
 >>> LINE 015: i = 1
 >>> LINE 020: odd = 1
 >>> LINE 015: i = 2
 >>> LINE 017: prime = 2
 >>> LINE 015: i = 3
 >>> LINE 021: prime = 3
 >>> LINE 015: i = 4
 >>> LINE 018: even = 4
 >>> LINE 015: i = 5
 >>> LINE 021: prime = 5
 >>> LINE 015: i = 6
 >>> LINE 018: even = 6
 >>> LINE 015: i = 7
 >>> LINE 021: prime = 7
 >>> LINE 015: i = 8
 >>> LINE 018: even = 8
 >>> LINE 015: i = 9
 >>> LINE 020: odd = 9
 >>> LINE 015: i = 10
 >>> LINE 018: even = 10
 >>> LINE 015: i = 11
 >>> LINE 021: prime = 11
 >>> LINE 015: i = 12
 >>> LINE 018: even = 12
 >>> LINE 015: i = 13
 >>> LINE 015: i = 14
 >>> LINE 015: i = 15
 >>> LINE 015: i = 16

                96 statements executed.
                 0 runtime errors.
                0.03 seconds total execution time.
```

This version of class SelectExecutor is not very efficient. Method searchBranches() performs a linear search of the SELECT_BRANCH subtrees, and searchConstants() performs a linear search of each branch's constants. Performance becomes slower if the source statement has more branches and constants. It takes longer to find a matching branch that appears later in the statement than one that appears earlier.

An Optimized SELECT Executor

You can achieve better SELECT runtime performance by using "jump tables." [Listing 8-12](#) shows method execute() of such an optimized version of class SelectExecutor.

```
// Jump table cache: entry key is a SELECT node,
//                   entry value is the jump table.
// Jump table: entry key is a selection value,
//             entry value is the branch statement.
private static HashMap<ICodeNode, HashMap<Object, ICodeNode>> jumpCache =
    new HashMap<ICodeNode, HashMap<Object, ICodeNode>>();

/**
 * Execute SELECT statement.
 * @param node the root node of the statement.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    // Is there already an entry for this SELECT node in the
    // jump table cache? If not, create a jump table entry.
    HashMap<Object, ICodeNode> jumpTable = jumpCache.get(node);
    if (jumpTable == null) {
        jumpTable = createJumpTable(node);
        jumpCache.put(node, jumpTable);
    }

    // Get the SELECT node&apos;s children.
    ArrayList<ICodeNode> selectChildren = node.getChildren();
    ICodeNode exprNode = selectChildren.get(0);

    // Evaluate the SELECT expression.
    ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
    Object selectValue = expressionExecutor.execute(exprNode);

    // If there is a selection, execute the SELECT_BRANCH&apos;s statement.
    ICodeNode statementNode = jumpTable.get(selectValue);
    if (statementNode != null) {
        StatementExecutor statementExecutor = new StatementExecutor(this);
        statementExecutor.execute(statementNode);
    }

    ++executionCount;  // count the SELECT statement itself
    return null;
}
```

Method `execute()` looks up the SELECT node's jump table in the jump table cache `jumpCache` and creates one if the table doesn't already exist. Therefore, it creates a SELECT node's jump table only once at runtime and only when it is first needed. After executing the SELECT expression, the method uses the selection value as the key into the jump table to extract the corresponding branch statement to execute, if the statement exists.

For each SELECT node, method `createJumpTable()` creates a jump table as a hash map that maps each SELECT constant value to the corresponding SELECT branch statement. See .

```
/**
 * Create a jump table for a SELECT node.
 * @param node the SELECT node.
 * @return the jump table.
 */
private HashMap<Object, ICodeNode> createJumpTable(ICodeNode node)
{
    HashMap<Object, ICodeNode> jumpTable = new HashMap<Object, ICodeNode>();

    // Loop over children that are SELECT_BRANCH nodes.
    ArrayList<ICodeNode> selectChildren = node.getChildren();
    for (int i = 1; i < selectChildren.size(); ++i) {
        ICodeNode branchNode = selectChildren.get(i);
        ICodeNode constantsNode = branchNode.getChildren().get(0);
        ICodeNode statementNode = branchNode.getChildren().get(1);

        // Loop over the constants children of the branch&apos;s CONSTANTS_NODE.
        ArrayList<ICodeNode> constantsList = constantsNode.getChildren();
        for (ICodeNode constantNode : constantsList) {

            // Create a jump table entry.
            Object value = constantNode.getAttribute(VALUE);
            jumpTable.put(value, statementNode);
        }
    }

    return jumpTable;
}
```

By including such an optimizing pass on the intermediate code for SELECT parse trees, you now have the beginnings of a *three-pass* interpreter (parse, optimize, execute). There are possible optimizations for the other statement parse trees. The last chapter of this book will have more to say about optimizations for both the interpreter and the compiler.

In the next two chapters, you'll parse Pascal declarations and add type checking to the statement parsers.

Chapter 9

Parsing Declarations

In this chapter, you'll parse Pascal declarations. Since all the information from the declarations enters into the symbol table, this chapter also extends the work of Chapter 4.

Goals and Approach

This chapter has the following goals:

- Parsers in the front end for Pascal constant definitions, type definitions, variable declarations, and type specifications.
- Additions to the symbol table to contain type information.

The approach will be to develop new classes to represent Pascal type information in the `intermediate`, `intermediate.symtabimpl`, and `intermediate.typeimpl` packages. To verify your work, you'll extend the `CrossReferencer` utility class you developed in Chapter 4 to include type information.

The declarations of a programming language are often the most challenging statements to parse:

- The syntax of declaration statements can be difficult.
- Declarations often include recursive definitions.
- You need to keep track of diverse information.
- You need to enter many new items into the symbol table.
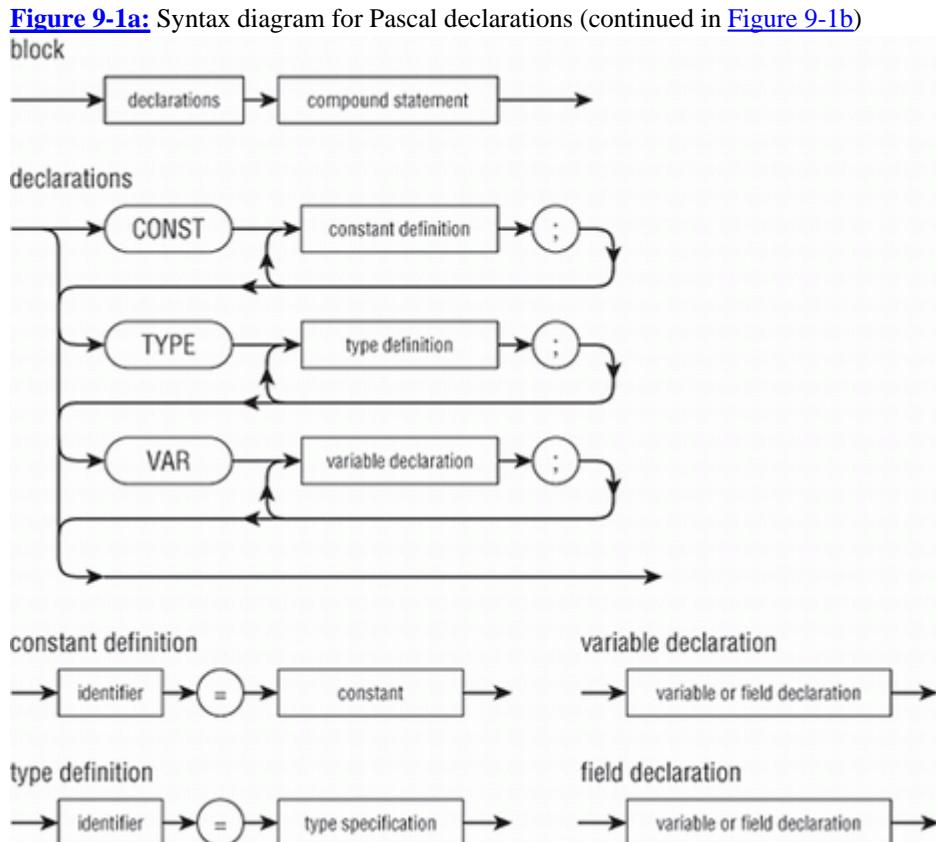
Pascal Declarations

Pascal declarations consist of four parts.[1] Each part is optional but must appear in this order: a constant definition part, a type definition part, a variable declaration part, and a procedure and function declaration part. This chapter is about the first three parts; you'll parse procedure and function declarations in Chapter 11. The syntax diagram in Figures

9-1a and 9-1b show that Pascal declarations have a relatively straightforward syntax.[2]

[1] You will not process Pascal's label declaration part since you did not process the `goto` control statement in Chapters 7 and 8.

[2] You will also not process Pascal's set and pointer types. Adding these types are straightforward "exercises for the reader."

Figure 9-1a: Syntax diagram for Pascal declarations (continued in Figure 9-1b)

Note that constants and types are *defined*, while variables, procedures, and functions are *declared.* We refer to the definition and declaration statements collectively as *declarations.*

A Pascal *constant definition* consists of an identifier followed by an = sign and a constant. An example constant definition part is:

```
CONST
    factor = 8;
    epsilon = 1.0e-6;
    ch = 'x';
    limit = -epsilon;
    message = 'Press the OK button to confirm your selection.';
```

A Pascal *type definition* consists of an identifier followed by an = sign and a *type specification.* A *simple type specification* is a previously defined type identifier, an enumeration type specification, or a subrange specification whose base type is an enumeration type or a non-real scalar type. An example type definition part is:
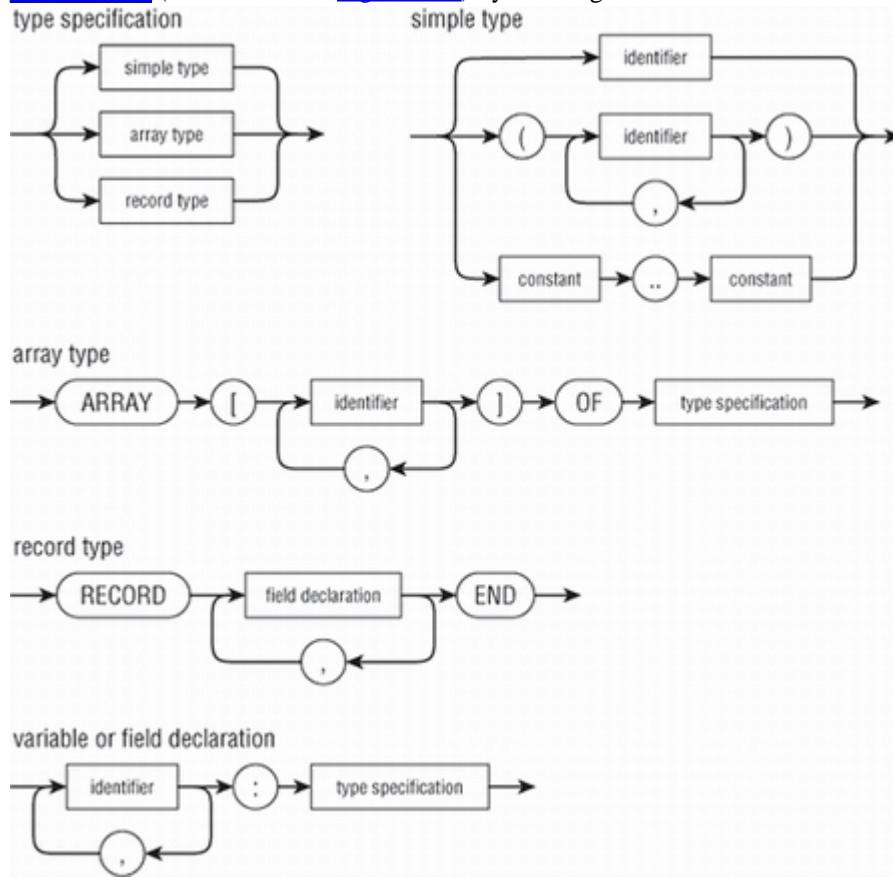
```
TYPE
    range1 = 0..factor; {subrange of integer}
    range2 = 'a'..'q';  {subrange of char}
    range3 = range1;    {type identifier}

    grades  = (A, B, C, D, F); {enumeration}
    passing = A..D;            {subrange of enumeration}

    week    = (monday, tuesday, wednesday, thursday, friday, saturday, sunday);
    weekday = monday..friday;
    weekend = saturday..sunday;
```

An *array type specification* has an index type and an element type. The index type must be a subrange or enumeration type, and the element type can be any Pascal type. Arrays can be multidimensional. Example array type definitions:

```
    ar1 = ARRAY [grades] OF integer;
    ar2 = ARRAY [(alpha, beta, gamma)] OF range2;
    ar3 = ARRAY [weekday] OF ar2;
    ar4 = ARRAY [range3] OF (foo bar, baz);
    ar5 = ARRAY [range1] OF ARRAY [range2] OF ARRAY[c..e] OF enum2;
    ar6 = ARRAY [range1, range2, c..e] OF enum2;
```

The array type definitions for `ar5` and `ar6` above show two equivalent ways to define a multidimensional array. Each dimension can be specified explicitly as in `ar5`, or they can be combined as in `ar6`.

A Pascal string type is an array of characters whose index type is an integer subrange with a lower limit of 1. Type `str` below is a string of length 10.

```
    str = ARRAY [1..10] OF char;
```

A Pascal *record type specification* consists of field declarations sandwiched between the keywords RECORD and END.[3] Each record field can be any Pascal type, including other record types. A colon separates a list of one or more field identifiers from the type. Two examples:

[3] A Pascal *record* is similar to a *structure* in other programming languages like C. You won't handle Pascal's variant records or WITH statements in this book.

```
    rec1 = RECORD
             i : integer;
             r : real;
             b1, b2 : boolean;
```

```
                    c : char
            END;

     rec2 = RECORD
                ten : integer;
                r : rec1;
                a1, a2, a3 : ARRAY [range3] OF range2;
            END;
```

Pascal variable declarations are syntactically similar to record field declarations. An example variable declaration part is:

```
VAR
    var1 : integer;
    var2, var3 : range2;
    var4 : ar2
    var5 : rec1;
    direction : (north, south, east, west);
```

Pascal supports *unnamed types*. In the definition of type `ar2` discussed previously, the index type is an unnamed enumeration type. Similarly, in the definitions of types `ar5` and `ar6`, the third dimension's index type is an unnamed subrange type. The type of the fields `a1`, `a2`, and `a3` of record type `rec2` is an unnamed array type. The type of variable `direction` is an unnamed enumeration type.

Types and the Symbol Table

The type specification parser that you'll write in this chapter will enter type information into the symbol table.

Design Note

Once again, as you did in Chapter 4, begin at the conceptual level. First, design language-independent interfaces that treat a type specification simply as a collection of attributes. Besides the attributes, a type specification has a form and it may have a type identifier. Afterwards, develop Pascal-specific implementations of these interfaces.
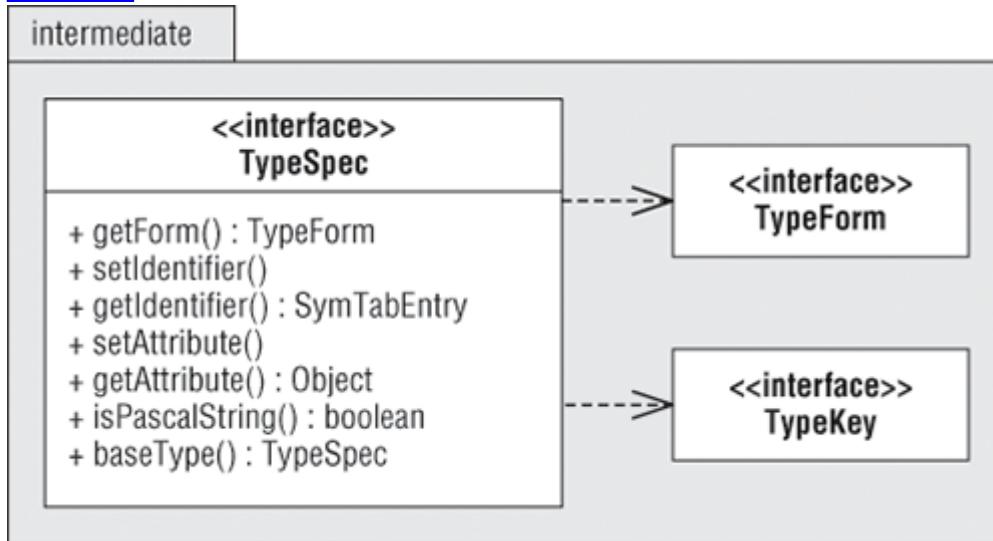

Type Specification Interfaces

[Figure 9-2](#) shows the UML diagram for the interfaces `TypeSpec`, `TypeForm`, and `TypeKey` in the `intermediate` package.

Method `getForm()` returns the form of the type, such as scalar, array, or record. Method `getIdentifier()` returns the symbol table entry of the type identifier or null if the type is unnamed. All other type specification information is stored and retrieved as named attribute values using methods `setAttribute()` and `getAttribute()`. Boolean method `isPascalString()` returns whether or not the type represents a Pascal string type. Method `baseType()` returns the base type of a subrange type; for other types, it simply returns the type itself.

[Listing 9-1](#) shows interface `TypeSpec`.

Figure 9-2: Interfaces `TypeSpec`, `TypeForm`, and `TypeKey`

Listing 9-1: Interface `TypeSpec`

```
package wci.intermediate;

/**
 * <h1>TypeSpec</h1>
 *
 * <p>The interface for a type specification.</p>
 */
public interface TypeSpec
{
    /**
     * Getter
     * @return the type form.
     */
    public TypeForm getForm();

    /**
     * Setter.
```

```
     * @param identifier the type identifier (symbol table entry).
     */
    public void setIdentifier(SymTabEntry identifier);

    /**
     * Getter.
     * @return the type identifier (symbol table entry).
     */
    public SymTabEntry getIdentifier();

    /**
     * Set an attribute of the specification.
     * @param key the attribute key.
     * @param value the attribute value.
     */
    public void setAttribute(TypeKey key, Object value);

    /**
     * Get the value of an attribute of the specification.
     * @param key the attribute key.
     * @return the attribute value.
     */
    public Object getAttribute(TypeKey key);

    /**
     * @return true if this is a Pascal string type.
     */
    public boolean isPascalString();

    /**
     * @return the base type of this type.
     */
    public TypeSpec baseType();
}
```

[Listing 9-2](#) shows the marker interface `TypeForm`.

[Listing 9-2:](#) Interface `TypeForm`

```
package wci.intermediate;

public interface TypeForm
{
}
```

[Listing 9-3](#) shows the marker interface `TypeKey` for the attribute keys.

[Listing 9-3:](#) Interface `TypeKey`

```
package wci.intermediate;

public interface TypeKey
{
}
```

Pascal Type Specification Implementation

As mentioned above, a Pascal type can be in the form of a scalar, an enumeration, a subrange, an array, or a record. The following table shows what type specification information the parser needs to extract for each type form.

Type form	Type specification information
Scalar	None
Enumeration	List of enumeration constant identifiers
Subrange	Base type
	Minimum value
	Maximum value
Array	Index type
	Minimum index value
	Maximum index value
	Element type
	Element count
Record	A separate symbol table for the field identifiers

Store the type specification information as attributes.

[Listing 9-4](#) shows the enumerated type `TypeFormImpl` in package `intermediate.typeimpl`, the Pascal-specific implementation of the interface `TypeForm`. Its enumerated values correspond to the first column of the table.

[Listing 9-4:](#) Class `TypeFormImpl`

```
package wci.intermediate.typeimpl;

import wci.intermediate.TypeForm;

/**
 * <h1>TypeFormImpl</h1>
 *
 * <p>Type forms for a Pascal type specification.</p>
 */
public enum TypeFormImpl implements TypeForm
{
    SCALAR, ENUMERATION, SUBRANGE, ARRAY, RECORD;
```

```
        public String toString()
        {
            return super.toString().toLowerCase();
        }
    }
}
```

Listing 9-5 shows the enumerated type `TypeKeyImpl` in package `intermediate.typeimpl`, the Pascal-specific implementation of the interface `TypeKey`. The enumerated values correspond to the second column of the table.

Listing 9-5: Enumerated type `TypeKeyImpl`

```
package wci.intermediate.typeimpl;

import wci.intermediate.TypeKey;

/**
 * <h1>TypeKeyImpl</h1>
 *
 * <p>Attribute keys for a Pascal type specification.</p>
 */
public enum TypeKeyImpl implements TypeKey
{
    // Enumeration
    ENUMERATION_CONSTANTS,

    // Subrange
    SUBRANGE_BASE_TYPE, SUBRANGE_MIN_VALUE, SUBRANGE_MAX_VALUE,

    // Array
    ARRAY_INDEX_TYPE, ARRAY_ELEMENT_TYPE, ARRAY_ELEMENT_COUNT,

    // Record
    RECORD_SYMTAB
}
```

As you did previously with the symbol table entries (Chapter 4) and the parse tree nodes (Chapter 5), implement each type specification as a hash map to gain flexibility. Listing 9-6 shows the key methods of class `TypeSpecImpl` in package `intermediate.typeimpl`, the Pascal-specific implementation of interface `TypeSpec`.

Listing 9-6: Key methods of class `TypeSpecImpl`

```
package wci.intermediate.typeimpl;

import java.util.HashMap;

import wci.intermediate.*;
import wci.intermediate.symtabimpl.Predefined;

import static wci.intermediate.typeimpl.TypeFormImpl.ARRAY;
import static wci.intermediate.typeimpl.TypeFormImpl.SUBRANGE;
import static wci.intermediate.typeimpl.TypeKeyImpl.*;

/**
 * <h1>TypeSpecImpl</h1>
 *
 * <p>A Pascal type specification implementation.</p>
 */
public class TypeSpecImpl
    extends HashMap<TypeKey, Object>
    implements TypeSpec
{
    private TypeForm form;          // type form
    private SymTabEntry identifier; // type identifier

    /**
     * Constructor.
     * @param form the type form.
     */
    public TypeSpecImpl(TypeForm form)
    {
        this.form = form;
        this.identifier = null;
    }

    /**
     * Constructor.
     * @param value a string value.
     */
    public TypeSpecImpl(String value)
    {
        this.form = ARRAY;

        TypeSpec indexType = new TypeSpecImpl(SUBRANGE);
        indexType.setAttribute(SUBRANGE_BASE_TYPE, Predefined.integerType);
        indexType.setAttribute(SUBRANGE_MIN_VALUE, 1);
        indexType.setAttribute(SUBRANGE_MAX_VALUE, value.length());

        setAttribute(ARRAY_INDEX_TYPE, indexType);
        setAttribute(ARRAY_ELEMENT_TYPE, Predefined.charType);
        setAttribute(ARRAY_ELEMENT_COUNT, value.length());
    }

    /**
     * Set an attribute of the specification.
     * @param key the attribute key.
     * @param value the attribute value.
     */
    public void setAttribute(TypeKey key, Object value)
    {
        this.put(key, value);
    }

    /**
     * Get the value of an attribute of the specification.
     * @param key the attribute key.
     * @return the attribute value.
     */
    public Object getAttribute(TypeKey key)
    {
        return this.get(key);
    }

    /**
```

```
      * @return true if this is a Pascal string type.
      */
    public boolean isPascalString()
    {
        if (form == ARRAY) {
            TypeSpec elmtType  = (TypeSpec) getAttribute(ARRAY_ELEMENT_TYPE);
            TypeSpec indexType = (TypeSpec) getAttribute(ARRAY_INDEX_TYPE);

            return (elmtType.baseType()  == Predefined.charType) &&
                   (indexType.baseType() == Predefined.integerType);
        }
        else {
            return false;
        }
    }

    /**
     * @return the base type of this type.
     */
    public Object getAttribute(TypeKey key)
    {
        return this.get(key);
    }

    /**
     * @return the base type of this type.
     */
    public TypeSpec baseType()
    {
        return (TypeFormImpl) form == SUBRANGE
                   ? (TypeSpec) getAttribute(SUBRANGE_BASE_TYPE)
                   : this;
    }
}
```

There are two constructors. The first simply creates an "empty" `TypeSpec` object given a form. The second constructor creates a `TypeSpec` object for a string constant. As explained at the beginning of this chapter, a Pascal string type is an array of characters with a subrange index type, which method `isPascalString()` verifies. This constructor creates the character array with an integer subrange index type whose values range from 1 up to and including the string length.

Figure 9-3 shows the relationships among symbol table entries and `TypeSpec` objects for several sample Pascal type definitions. Important items to note include:

- A type identifier's symbol table entry points to the corresponding `TypeSpec` object which in turn points back to the symbol table entry. However, a type specification for an unnamed type (such as the unnamed array index type in the figure) has no pointer to a symbol table entry.
- An enumeration type specification's `ENUMERATION_CONSTANTS` attribute is an array list of the symbol table entries of the constant identifiers.
- A subrange type specification has `SUBRANGE_MIN_VALUE` and `SUBRANGE_MAX_VALUE` attributes, and its `SUBRANGE_BASE_TYPE` attribute is the subrange's base type specification.
- An array type specification's `ARRAY_INDEX_TYPE` and `ARRAY_ELEMENT_TYPE` attributes are the type specifications for the array's indexes and elements, respectively.
- A record type specification's `RECORD_SYMTAB` attribute is the symbol table containing entries for the record type's field identifiers.

A Type Factory

Listing 9-7 shows class `TypeFactory` in package `intermediate`, which creates `TypeSpec` objects. It has two static methods, one for each `TypeSpecImpl` constructor.

Design Note

Again use the Factory Method Design Pattern, this time to keep clients of type specifications loosely coupled from specific type specification implementations.

Listing 9-7: Factory class `TypeFactory`

```
package wci.intermediate;

import wci.intermediate.typeimpl.*;

/**
 * <h1>TypeFactory</h1>
 *
 * <p>A factory for creating type specifications.</p>
 */
public class TypeFactory
{
    /**
     * Create a type specification of a given form.
     * @param form the form.
     * @return the type specification.
     */
    public static TypeSpec createType(TypeForm form)
    {
        return new TypeSpecImpl(form);
    }

    /**
     * Create a string type specification.
     * @param value the string value.
     * @return the type specification.
     */
    public static TypeSpec createStringType(String value)
    {
        return new TypeSpecImpl(value);
    }
```
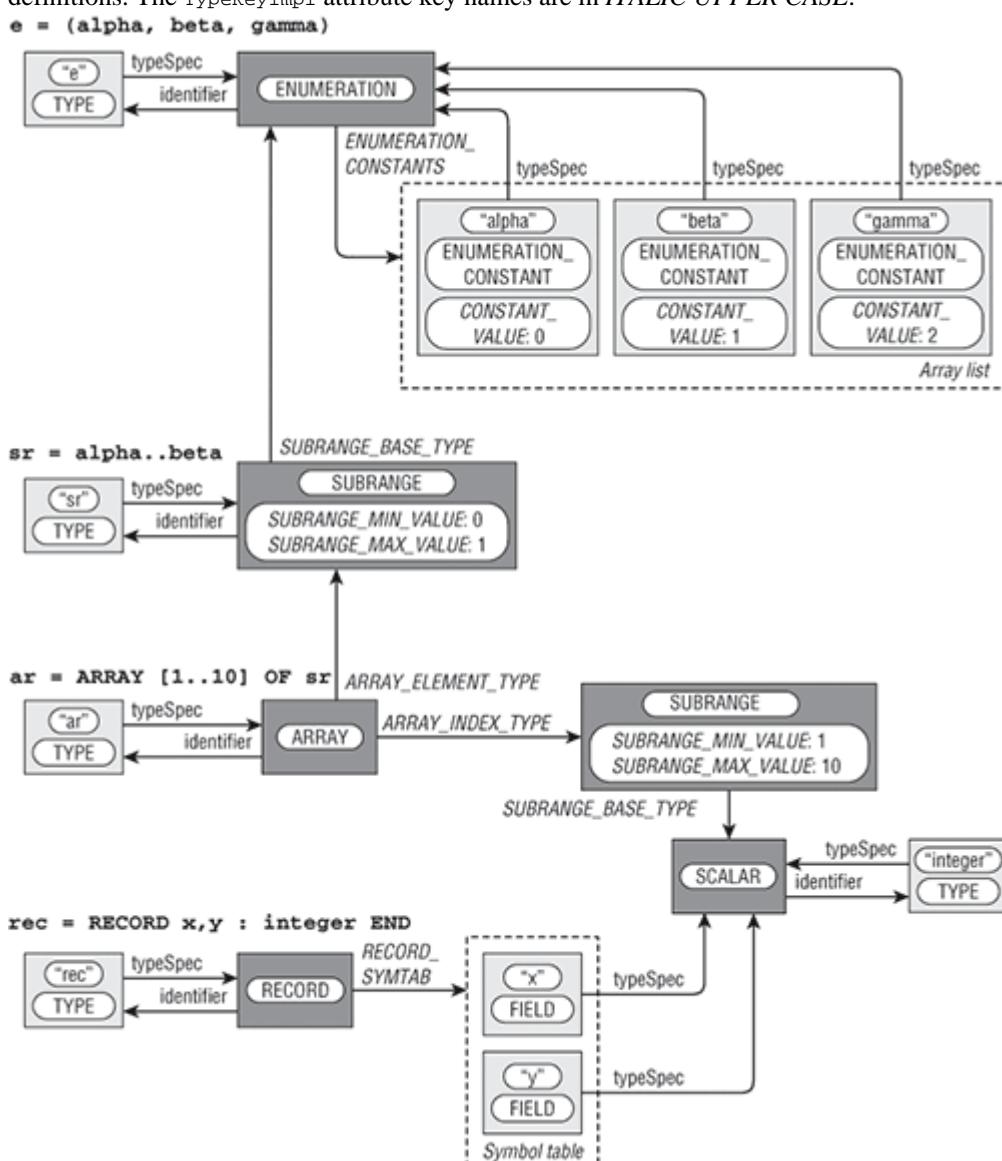
Scope and the Symbol Table Stack

Scope refers to the part of the source program where certain identifiers can be used (in other words, everywhere in the program where the definitions of those identifiers are in effect). We also say that each and every identifier *has a scope* or *belongs to a scope*. Scope is closely related to the concepts of nesting levels and the symbol table stack, briefly introduced in Chapter 4.

Pascal has a number of identifiers that name predefined elements of the language, such as the types `integer`, `real`, `boolean`, and `char` and the constants `true` and `false`. In a later chapter, you'll encounter identifiers that name predefined routines like `writeln`. By default, the scope of these predefined *global identifiers* is the entire program. They are implicitly defined at scope nesting level 0, the *global scope*. The program name is also defined at this level.

Until this chapter, you've only had a single symbol table on the symbol table stack at nesting level 0, so you've treated all the identifiers as if they were all global. Actually, all the identifiers explicitly defined by the programmer at the top level within a program (the names of constants, types, variables, and routines) are at nesting level 1, the *program scope*.

Each Pascal record definition creates a new scope for its field identifiers. The nesting level of this new scope is one higher than the nesting level of the scope in which the record itself is defined. Nested record definitions create scopes at even higher nesting levels. Similarly, as you'll see in Chapter 11, each declared Pascal procedure or function creates a new scope at the nesting level that corresponds to how deeply the routine is nested within other routines.

Figure 9-3: The relationships among `SymTabEntry` objects (light gray) and `TypeSpec` objects (dark gray) for various sample type definitions. The `TypeKeyImpl` attribute key names are in *ITALIC UPPER CASE*.

An identifier can be redefined within a nested scope. For example, a program may define identifier `str` to be the name of a string type. But the program may also have a record definition that defines `str` to be the name of a field. Inside of the record's nested scope, this redefinition of `str` as a field is valid, and the original definition of `str` out in the program scope is hidden. But outside of the record, back within the program scope, the definition of `str` as a string type applies.

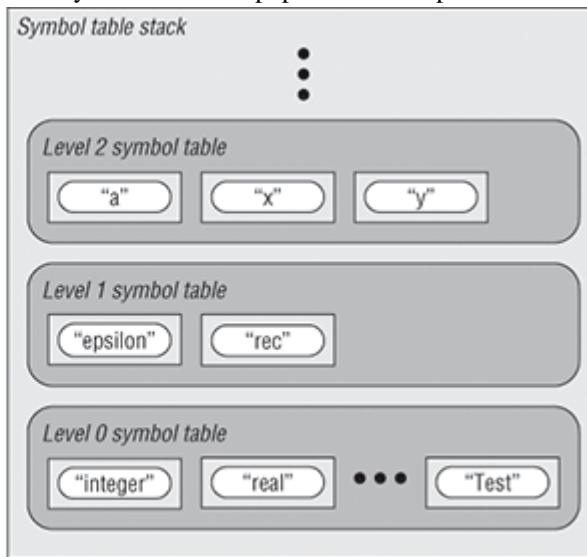This discussion raises two important issues:

- Each scope must have its own separate symbol table. There will be a symbol table at nesting level 0 for the global identifiers. The program scope will have a symbol table at level 1 for its identifiers. Each record definition will have a symbol table for its field identifiers. Each routine will have a separate symbol table for its identifiers.

- As the parser parses a program source from top to bottom, it will enter and exit nested scopes, and so these scopes will be maintained on the symbol table stack. Each time the parser enters a scope it must push the scope's symbol table onto the stack and increase the nesting level by 1. When the parser exits the scope, it must pop the scope's symbol table off the stack and decrease the nesting level by 1. The symbol table stack will grow as the parser enters more deeply nested scopes.

Figure 9-4 illustrates the concept of a symbol table stack.

Figure 9-4: The symbol table stack just before the parser is finished with the RECORD type. The predefined global identifiers such as integer and real and the program name Test are in the level 0 symbol table. The identifiers defined in the program scope such as epsilon and rec are in the level 1 symbol table. The RECORD type specification pushes a new symbol table at level 2 onto the stack to contain its field identifiers a, x, and y. This symbol table will pop off once the parser is done with the RECORD type.

You've had interface SymTabStack in the intermediate package since Chapter 4. Now add several new methods to push and pop symbol tables and to set and get the program identifier, as shown in Listing 9-8.

Listing 9-8: New methods defined in interface **SymTabStack**

```
/**
 * Setter.
 * @param entry the symbol table entry for the main program identifier.
 */
public void setProgramId(SymTabEntry entry);

/**
 * Getter.
 * @return the symbol table entry for the main program identifier.
 */
public SymTabEntry getProgramId();

/**
 * Push a new symbol table onto the stack.
 * @return the pushed symbol table.
 */
public SymTab push();

/**
 * Push a symbol table onto the stack.
 * @param symTab the symbol table to push.
 * @return the pushed symbol table.
 */
public SymTab push(SymTab symTab);

/**
 * Pop a symbol table off the stack.
 * @return the popped symbol table.
 */
public SymTab pop();
```

Methods setProgramId() and getProgramId() are for the symbol table entry of the main program identifier.

Methods push() and pop() will push and pop a symbol table onto and off the symbol table stack, respectively. There are two push() methods; one will create a new symbol table to push, and the other will push an existing symbol table.

Class SymTabStackImpl in package intermediate.symtabimpl must now include a new field:

```
private SymTabEntry programId;    // entry for the main program id
```

Listing 9-9 shows the implementation of the new push() and pop() methods in class SymTabStackImpl and a new version of method lookup().

Listing 9-9: Methods **push(), pop(), and lookup()** in class **SymTabStackImpl**

```
/**
 * Push a new symbol table onto the symbol table stack.
 * @return the pushed symbol table.
 */
public SymTab push()
{
    SymTab symTab = SymTabFactory.createSymTab(++currentNestingLevel);
    add(symTab);

    return symTab;
}

/**
 * Push a symbol table onto the symbol table stack.
 * @return the pushed symbol table.
```

```
    */
    public SymTab push(SymTab symTab)
    {
        ++currentNestingLevel;
        add(symTab);

        return symTab;
    }

    /**
     * Pop a symbol table off the symbol table stack.
     * @return the popped symbol table.
     */
    public SymTab pop()
    {
        SymTab symTab = get(currentNestingLevel);
        remove(currentNestingLevel--);

        return symTab;
    }

    /**
     * Look up an existing symbol table entry throughout the stack.
     * @param name the name of the entry.
     * @return the entry, or null if it does not exist.
     */
    public SymTabEntry lookup(String name)
    {
        SymTabEntry foundEntry = null;

        // Search the current and enclosing scopes.
        for (int i = currentNestingLevel; (i >= 0) && (foundEntry == null); --i)
        {
            foundEntry = get(i).lookup(name);
        }

        return foundEntry;
    }
```

Both versions of method `push()` increment the current nesting level, and method `pop()` decrements it.

Method `lookup()` now searches the current scope and, if necessary, the enclosing scopes. It starts its search for an identifier in the local symbol table at the top of the stack and works its way down the stack until it encounters the first symbol table that contains the identifier, or else it doesn't find the identifier at all. In other words, the method first looks for the identifier in the current scope, and if necessary, the method searches the enclosing scopes outwards one by one.

How Identifiers Are Defined

You can no longer simply assume that an identifier is a variable. An identifier's symbol table entry must indicate how it is defined, as a constant or a type or a record field or a variable, etc. An identifier can also have a type, which also must be indicated by its symbol table entry. Therefore, add new setter and getter methods to interface `SymTabEntry` in the `intermediate` package. See Listing 9-10.

Listing 9-10: New setter and getter methods in interface `SymTabEntry`

```
    /**
     * Setter.
     * @param definition the definition to set.
     */
    public void setDefinition(Definition definition);

    /**
     * Getter.
     * @return the definition.
     */
    public Definition getDefinition();

    /**
     * Setter.
     * @param typeSpec the type specification to set.
     */
    public void setTypeSpec(TypeSpec typeSpec);

    /**
     * Getter.
     * @return the type specification.
     */
    public TypeSpec getTypeSpec();
```

These new methods are implemented straightforwardly in class `SymTabEntryImpl` in the `intermediate.symtabimpl` package, which now must include two new fields:

```
    private Definition definition;          // how the identifier is defined
    private TypeSpec typeSpec;              // type specification
```

Listing 9-11 shows the new `Definition` interface.

Listing 9-11: Interface `Definition`

```
package wci.intermediate;

/**
 * <h1>Definition</h1>
 *
 * <p>The interface for how a symbol table entry is defined.</p>
 */
public interface Definition
{
    /**
     * Getter.
     * @return String the text of the definition.
     */
    public String getText();
}
```

Listing 9-12 shows the enumerated type `DefinitionImpl` in the `intermediate.symtabimpl` package, which implements the `Definition` interface. Its enumerated values represent all the ways an identifier can be defined.

Listing 9-12: Class `DefinitionImpl`

```
package wci.intermediate.symtabimpl;

import wci.intermediate.Definition;

/**
 * <h1>DefinitionImpl</h1>
 *
 * <p>How a Pascal symbol table entry is defined.</p>
 */
public enum DefinitionImpl implements Definition
{
    CONSTANT, ENUMERATION_CONSTANT("enumeration constant"),
    TYPE, VARIABLE, FIELD("record field"),
    VALUE_PARM("value parameter"), VAR_PARM("VAR parameter"),
    PROGRAM_PARM("program parameter"),
    PROGRAM, PROCEDURE, FUNCTION,
    UNDEFINED;

    private String text;

    /**
     * Constructor.
     */
    DefinitionImpl()
    {
        this.text = this.toString().toLowerCase();
    }

    /**
     * Constructor.
     * @param text the text for the definition code.
     */
    DefinitionImpl(String text)
    {
        this.text = text;
    }

    /**
     * Getter.
     * @return String the text of the definition code.
     */
    public String getText()
    {
        return text;
    }
}
```

Predefined Types and Constants

Listing 9-13 shows class `Predefined`. Its static fields represent Pascal's predefined integer, real, boolean, and character types along with the undefined type, and the predefined identifiers `integer`, `real`, `boolean`, `char`, `false`, and `true`.

Listing 9-13: Class `Predefined`

```
package wci.intermediate.symtabimpl;

import java.util.ArrayList;

import wci.intermediate.*;
import wci.intermediate.symtabimpl.*;

import static wci.intermediate.symtabimpl.SymTabKeyImpl.*;
import static wci.intermediate.typeimpl.TypeFormImpl.*;
import static wci.intermediate.typeimpl.TypeKeyImpl.*;

/**
 * <h1>Predefined</h1>
 *
 * <p>Enter the predefined Pascal types, identifiers, and constants
 * into the symbol table.</p>
 */
public class Predefined
{
    // Predefined types.
    public static TypeSpec integerType;
    public static TypeSpec realType;
    public static TypeSpec booleanType;
    public static TypeSpec charType;
    public static TypeSpec undefinedType;

    // Predefined identifiers.
    public static SymTabEntry integerId;
    public static SymTabEntry realId;
    public static SymTabEntry booleanId;
    public static SymTabEntry charId;
    public static SymTabEntry falseId;
    public static SymTabEntry trueId;

    /**
     * Initialize a symbol table stack with predefined identifiers.
     * @param symTab the symbol table stack to initialize.
     */
    public static void initialize(SymTabStack symTabStack)
    {
        initializeTypes(symTabStack);
        initializeConstants(symTabStack);
    }

    /**
```

```
    * Initialize the predefined types.
    * @param symTabStack the symbol table stack to initialize.
    */
   private static void initializeTypes(SymTabStack symTabStack)
   {
       // Type integer.
       integerId = symTabStack.enterLocal("integer");
       integerType = TypeFactory.createType(SCALAR);
       integerType.setIdentifier(integerId);
       integerId.setDefinition(DefinitionImpl.TYPE);
       integerId.setTypeSpec(integerType);

       // Type real.
       realId = symTabStack.enterLocal("real");
       realType = TypeFactory.createType(SCALAR);
       realType.setIdentifier(realId);
       realId.setDefinition(DefinitionImpl.TYPE);
       realId.setTypeSpec(realType);

       // Type boolean.
       booleanId = symTabStack.enterLocal("boolean");
       booleanType = TypeFactory.createType(ENUMERATION);
       booleanType.setIdentifier(booleanId);
       booleanId.setDefinition(DefinitionImpl.TYPE);
       booleanId.setTypeSpec(booleanType);

       // Type char.
       charId = symTabStack.enterLocal("char");
       charType = TypeFactory.createType(SCALAR);
       charType.setIdentifier(charId);
       charId.setDefinition(DefinitionImpl.TYPE);
       charId.setTypeSpec(charType);

       // Undefined type.
       undefinedType = TypeFactory.createType(SCALAR);
   }

   /**
    * Initialize the predefined constant.
    * @param symTabStack the symbol table stack to initialize.
    */
   private static void initializeConstants(SymTabStack symTabStack)
   {
       // Boolean enumeration constant false.
       falseId = symTabStack.enterLocal("false");
       falseId.setDefinition(DefinitionImpl.ENUMERATION_CONSTANT);
       falseId.setTypeSpec(booleanType);
       falseId.setAttribute(CONSTANT_VALUE, new Integer(0));

       // Boolean enumeration constant true.
       trueId = symTabStack.enterLocal("true");
       trueId.setDefinition(DefinitionImpl.ENUMERATION_CONSTANT);
       trueId.setTypeSpec(booleanType);
       trueId.setAttribute(CONSTANT_VALUE, new Integer(1));

       // Add false and true to the boolean enumeration type.
       ArrayList<SymTabEntry> constants = new ArrayList<SymTabEntry>();
       constants.add(falseId);
       constants.add(trueId);
       booleanType.setAttribute(ENUMERATION_CONSTANTS, constants);
   }
}
```

The public static method `initialize()` initializes these predefined types and identifiers. You'll see later that the parser calls this method at nesting level 0 when it first starts parsing a program.

Private method `initializeTypes()` creates each predefined type's specification and identifier. It uses the type factory to create the `TypeSpec` object. It creates the symbol table entry for the type identifier and sets its definition and type specification.

Private method `initializeConstants()` creates the symbol table entries for the predefined boolean constants `false` and `true` and sets each entry's definition and type specification and the CONSTANT_VALUE attribute value (0 for `false` and 1 for `true`). The method sets the `boolean` type specification's ENUMERATION_CONSTANTS attribute value to the list of these symbol table entries.

Parsing Pascal Declarations

You're ready to move to the front end and parse the Pascal declarations.
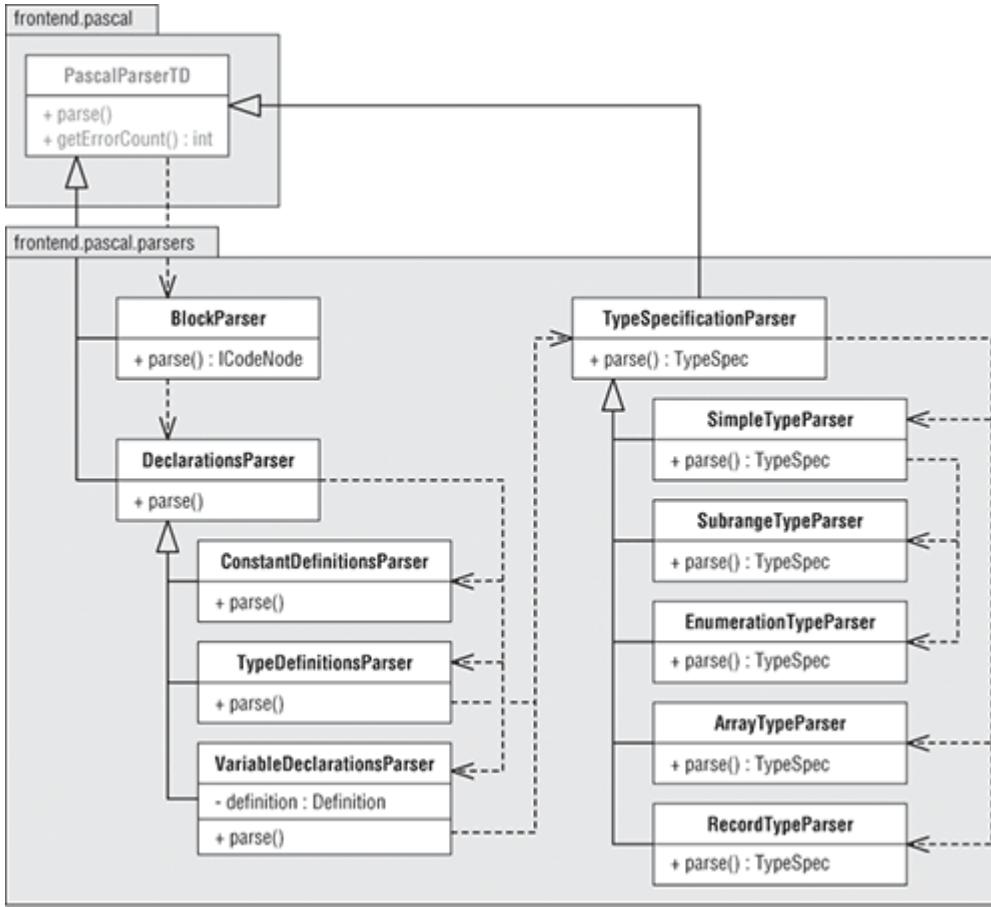
Figure 9-5 shows the UML class diagram of the declarations parsing classes.

Class `BlockParser`, which parses a Pascal block, is a subclass of `PascalParserTD`. As shown in the syntax diagram in Figure 9-1, a block consists of declarations followed by a compound statement, and so `BlockParser` depends on class `DeclarationsParser`, which is another subclass of `PascalParserTD`. Class `DeclarationsParser` has subclasses `ConstantsDefinitionParser`, `TypeDefinitionsParser`, and `VariableDeclarationsParser`, and it depends on each one of them.

Yet another subclass of `PascalParserTD` is `TypeSpecificationParser`, which in turn has subclasses `SimpleTypeParser`, `SubrangeTypeParser`, `EnumerationTypeParser`, `ArrayTypeParser`, and `RecordTypeParser`. These subclasses correspond to the components in the syntax diagram. Each subclass has a `parse()` method that returns a `TypeSpec` object. `TypeSpecificationParser` depends on `SimpleTypeParser`, `ArrayTypeParser`, and `RecordTypeParser`. `SimpleTypeParser` in turn depends on `SubrangeTypeParser` and `EnumerationTypeParser`.

Both `TypeDefinitionsParser` and `VariableDeclarationsParser` depend on `TypeSpecificationParser` since type definitions and variable declarations each contains a type specification.

Figure 9-5: The classes for parsing Pascal declarations

Begin by updating class `PascalParserTD`. See [Listing 9-14](#).

Listing 9-14:: **Method `parse()` of class `PascalParserTD`**

```java
private SymTabEntry routineId;  // name of the routine being parsed

/**
 * Parse a Pascal source program and generate the symbol table
 * and the intermediate code.
 * @throws Exception if an error occurred.
 */
public void parse()
    throws Exception
{
    long startTime = System.currentTimeMillis();

    ICode iCode = ICodeFactory.createICode();
    Predefined.initialize(symTabStack));

    Predefined.initialize(symTabStack);

    // Create a dummy program identifier symbol table entry.
    routineId = symTabStack.enterLocal("DummyProgramName".toLowerCase());
    routineId.setDefinition(DefinitionImpl.PROGRAM);
    symTabStack.setProgramId(routineId);

    // Push a new symbol table onto the symbol table stack and set
    // the routine&apos;s symbol table and intermediate code.
    routineId.setAttribute(ROUTINE_SYMTAB, symTabStack.push());
    routineId.setAttribute(ROUTINE_ICODE, iCode);

    BlockParser blockParser = new BlockParser(this);

    try {
        Token token = nextToken();

        // Parse a block.
        ICodeNode rootNode = blockParser.parse(token, routineId);
        iCode.setRoot(rootNode);
        symTabStack.pop();

        // Look for the final period.
        token = currentToken();
        if (token.getType() != DOT) {
            errorHandler.flag(token, MISSING_PERIOD, this);
        }
        token = currentToken();

        // Send the parser summary message.
        float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
        sendMessage(new Message(PARSER_SUMMARY,
                                new Number[] {token.getLineNumber(),
                                              getErrorCount(),
                                              elapsedTime}));
    }
    catch (java.io.IOException ex) {
        errorHandler.abortTranslation(IO_ERROR, this);
    }
}
```

The new field `routineId` refers to the symbol table entry of the routine being parsed. For now, this will be the name identifier of the main program, albeit a dummy name. Later, this will be the actual program identifier, or the procedure or function name identifier.

Method `parse()` calls `Predefined.initialize()` to initialize the predefined types and constants. It creates a symbol table entry for the program identifier (`"DummyProgramName"` for now). All of this happens at nesting level 0. In particular, the method enters the program identifier into the level 0 symbol table.[4]

[4] You'll see later that a procedure or function name identifier is always defined in the scope that immediately encloses the scope that the routine creates for its own identifiers. In other words, if the procedure or function name identifier is defined at level *n*, the identifiers defined within the procedure or function are at level *n*+1. A record type name identifier is treated similarly.

The first of the two statements

```
routineId.setAttribute(ROUTINE_SYMTAB, symTabStack.push());
routineId.setAttribute(ROUTINE_ICODE, iCode);
```

sets the `ROUTINE_SYMTAB` symbol table attribute value of the program identifier's symbol table entry. This is a new symbol table that the call to `symTabStack.push()` pushes onto the symbol table stack at nesting level 1. The second statement sets the `ROUTINE_ICODE` attribute value to the intermediate code. Each program, procedure, and function will have its own symbol table and its own intermediate code.

Because you're now storing a reference to each routine's intermediate code in the symbol table entry of the routine's name, remove all references to the protected field `iCode` from the framework `Parser` class.

The new Pascal parser subclass `BlockParser` in package `frontend.pascal.parsers` is responsible for parsing the program block and later, each procedure and function block. Listing 9-15 shows its `parse()` method, which calls `declarationsParser.parse()` to parse a block's declarations and `statementParser.parse()` to parse the block's compound statement. Besides the current token, it is passed the symbol table entry of the name of the routine being parsed.

Listing 9-15: Method `parse()` of class `BlockParser`

```
/**
 * Parse a block.
 * @param token the initial token.
 * @param routineId the symbol table entry of the routine name.
 * @return the root node of the parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token, SymTabEntry routineId)
    throws Exception
{
    DeclarationsParser declarationsParser = new DeclarationsParser(this);
    StatementParser statementParser = new StatementParser(this);

    // Parse any declarations.
    declarationsParser.parse(token);

    token = synchronize(StatementParser.STMT_START_SET);
    TokenType tokenType = token.getType();
    ICodeNode rootNode = null;

    // Look for the BEGIN token to parse a compound statement.
    if (tokenType == BEGIN) {
        rootNode = statementParser.parse(token);
    }

    // Missing BEGIN: Attempt to parse anyway if possible.
    else {
        errorHandler.flag(token, MISSING_BEGIN, this);

        if (StatementParser.STMT_START_SET.contains(tokenType)) {
            rootNode = ICodeFactory.createICodeNode(COMPOUND);
            statementParser.parseList(token, rootNode, END, MISSING_END);
        }
    }

    return rootNode;
}
```

Listing 9-16 shows the `parse()` method of the Pascal parser subclass `DeclarationsParser`. The method successively looks for the keywords `CONST`, `TYPE`, and `VAR`. If it finds each keyword, it calls `constantDefinitionsParser.parse()`, `typeDefinitionsParser.parse()`, and `variableDeclarationsParser.parse()`, respectively. It sets the `definition` of `variableDeclarationsParser` to `VARIABLE`.

Listing 9-16: Method `parse()` of class `DeclarationsParser`

```
static final EnumSet<PascalTokenType> DECLARATION_START_SET =
    EnumSet.of(CONST, TYPE, VAR, PROCEDURE, FUNCTION, BEGIN);

static final EnumSet<PascalTokenType> TYPE_START_SET =
    DECLARATION_START_SET.clone();
static {
    TYPE_START_SET.remove(CONST);
}

static final EnumSet<PascalTokenType> VAR_START_SET =
    TYPE_START_SET.clone();
static {
    VAR_START_SET.remove(TYPE);
}

static final EnumSet<PascalTokenType> ROUTINE_START_SET =
    VAR_START_SET.clone();
static {
    ROUTINE_START_SET.remove(VAR);
}

/**
 * Parse declarations.
 * To be overridden by the specialized declarations parser subclasses.
 * @param token the initial token.
 * @throws Exception if an error occurred.
 */
public void parse(Token token)
    throws Exception
{
    token = synchronize(DECLARATION_START_SET);

    if (token.getType() == CONST) {
        token = nextToken();  // consume CONST

        ConstantDefinitionsParser constantDefinitionsParser =
```

```
            new ConstantDefinitionsParser(this);
        constantDefinitionsParser.parse(token);
    }

    token = synchronize(TYPE_START_SET);

    if (token.getType() == TYPE) {
        token = nextToken();  // consume TYPE

        TypeDefinitionsParser typeDefinitionsParser =
            new TypeDefinitionsParser(this);
        typeDefinitionsParser.parse(token);
    }

    token = synchronize(VAR_START_SET);

    if (token.getType() == VAR) {
        token = nextToken();  // consume VAR

        VariableDeclarationsParser variableDeclarationsParser =
            new VariableDeclarationsParser(this);
        variableDeclarationsParser.setDefinition(VARIABLE);
        variableDeclarationsParser.parse(token);
    }

    token = synchronize(ROUTINE_START_SET);
}
```

Parsing Constant Definitions

The `parse()` method of the declarations parser subclass `ConstantDefinitionsParser` parses a sequence of constant definitions separated by semicolons. See [Listing 9-17](#). It calls `symTab.lookupLocal()` to search only the local symbol table to guard against redefining an identifier within the same scope. It enters each identifier into the local symbol table and calls the entry's `appendLineNumber()` to append the current source line number. You'll see a call to `appendLineNumber()` from now on whenever a new identifier (constant, type, enumeration, or variable) is defined.

[Listing 9-17:](#) Method `parse()` of class `ConstantDefinitionsParser`

```
    // Synchronization set for a constant identifier.
    private static final EnumSet<PascalTokenType> IDENTIFIER_SET =
        DeclarationsParser.TYPE_START_SET.clone();
    static {
        IDENTIFIER_SET.add(IDENTIFIER);
    }

    // Synchronization set for starting a constant.
    static final EnumSet<PascalTokenType> CONSTANT_START_SET =
        EnumSet.of(IDENTIFIER, INTEGER, REAL, PLUS, MINUS, STRING, SEMICOLON);

    // Synchronization set for the = token.
    private static final EnumSet<PascalTokenType> EQUALS_SET =
        CONSTANT_START_SET.clone();
    static {
        EQUALS_SET.add(EQUALS);
        EQUALS_SET.add(SEMICOLON);
    }

    // Synchronization set for the start of the next definition or declaration.
    private static final EnumSet<PascalTokenType> NEXT_START_SET =
        DeclarationsParser.TYPE_START_SET.clone();
    static {
        NEXT_START_SET.add(SEMICOLON);
        NEXT_START_SET.add(IDENTIFIER);
    }

    /**
     * Parse constant definitions.
     * @param token the initial token.
     * @throws Exception if an error occurred.
     */
    public void parse(Token token)
        throws Exception
    {
        token = synchronize(IDENTIFIER_SET);

        // Loop to parse a sequence of constant definitions
        // separated by semicolons.
        while (token.getType() == IDENTIFIER) {
            String name = token.getText().toLowerCase();
            SymTabEntry constantId = symTabStack.lookupLocal(name);

            // Enter the new identifier into the symbol table
            // but don&apos;t set how it&apos;s defined yet.
            if (constantId == null) {
                constantId = symTabStack.enterLocal(name);
                constantId.appendLineNumber(token.getLineNumber());
            }
            else {
                errorHandler.flag(token, IDENTIFIER_REDEFINED, this);
                constantId = null;
            }

            token = nextToken();  // consume the identifier token

            // Synchronize on the = token.
            token = synchronize(EQUALS_SET);
            if (token.getType() == EQUALS) {
                token = nextToken();  // consume the =
            }
            else {
                errorHandler.flag(token, MISSING_EQUALS, this);
            }

            // Parse the constant value.
            Token constantToken = token;
            Object value = parseConstant(token);

            // Set identifier to be a constant and set its value.
            if (constantId != null) {
                constantId.setDefinition(CONSTANT);
                constantId.setAttribute(CONSTANT_VALUE, value);

                // Set the constant&apos;s type.
                TypeSpec constantType =
                    constantToken.getType() == IDENTIFIER
```

```
                    ? getConstantType(constantToken)
                    : getConstantType(value);
                constantId.setTypeSpec(constantType);
            }

            token = currentToken();
            TokenType tokenType = token.getType();

            // Look for one or more semicolons after a definition.
            if (tokenType == SEMICOLON) {
                while (token.getType() == SEMICOLON) {
                    token = nextToken();  // consume the ;
                }
            }

            // If at the start of the next definition or declaration,
            // then missing a semicolon.
            else if (NEXT_START_SET.contains(tokenType)) {
                errorHandler.flag(token, MISSING_SEMICOLON, this);
            }

            token = synchronize(IDENTIFIER_SET);
        }
    }
```

Method `parse()` sets the entry's definition to `DefinitionImpl.CONSTANT` and the `CONSTANT_VALUE` attribute value to the constant's value. It calls `getConstantType()` to set the entry's type specification.

Design Note

Method `parse()` waits until it has successfulltype="general"y parsed the constant's value before updating identifier's symbol table entry in case there are any errors in parsing the value and to be able to flag erroneous definitions such as `pi = pi`.

Method `parseConstant()` parses and returns the constant's value, which can be a number, a string, or an identifier that was previously defined as a constant identifier. See Listing 9-18. It calls method `parseIdentifierConstant()` for the latter case, which also handles enumeration constant identifiers.

Listing 9-18: Method `parseConstant()` of class `ConstantDefinitionsParser`

```
/**
 * Parse a constant value.
 * @param token the current token.
 * @return the constant value.
 * @throws Exception if an error occurred.
 */
protected Object parseConstant(Token token)
    throws Exception
{
    TokenType sign = null;

    // Synchronize at the start of a constant.
    token = synchronize(CONSTANT_START_SET);
    TokenType tokenType = token.getType();

    // Plus or minus sign?
    if ((tokenType == PLUS) || (tokenType == MINUS)) {
        sign = tokenType;
        token = nextToken();  // consume sign
    }

    // Parse the constant.
    switch ((PascalTokenType) token.getType()) {

        case IDENTIFIER: {
            return parseIdentifierConstant(token, sign);
        }

        case INTEGER: {
            Integer value = (Integer) token.getValue();
            nextToken();  // consume the number
            return sign == MINUS ? -value : value;
        }

        case REAL: {
            Float value = (Float) token.getValue();
            nextToken();  // consume the number
            return sign == MINUS ? -value : value;
        }

        case STRING: {
            if (sign != null) {
                errorHandler.flag(token, INVALID_CONSTANT, this);
            }

            nextToken();  // consume the string
            return (String) token.getValue();
        }

        default: {
            errorHandler.flag(token, INVALID_CONSTANT, this);
            return null;
        }
    }
}
```

Listing 9-19 shows method `parseIdentifierConstant()`.

Listing 9-19: Method `parseIdentifierConstant()` of class `ConstantDefinitionsParser`

```
/**
 * Parse an identifier constant.
 * @param token the current token.
 * @param sign the sign, if any.
```

```
 * @return Object the constant value.
 * @throws Exception if an error occurred.
 */
protected Object parseIdentifierConstant(Token token, TokenType sign)
    throws Exception
{
    String name = token.getText();
    SymTabEntry id = symTabStack.lookup(name);

    nextToken();  // consume the identifier

    // The identifier must have already been defined
    // as an constant identifier.
    if (id == null) {
        errorHandler.flag(token, IDENTIFIER_UNDEFINED, this);
        return null;
    }

    Definition definition = id.getDefinition();

    if (definition == CONSTANT) {
        Object value = id.getAttribute(CONSTANT_VALUE);
        id.appendLineNumber(token.getLineNumber());

        if (value instanceof Integer) {
            return sign == MINUS ? -((Integer) value) : value;
        }
        else if (value instanceof Float) {
            return sign == MINUS ? -((Float) value) : value;
        }
        else if (value instanceof String) {
            if (sign != null) {
                errorHandler.flag(token, INVALID_CONSTANT, this);
            }

            return value;
        }
        else {
            return null;
        }
    }
    else if (definition == ENUMERATION_CONSTANT) {
        Object value = id.getAttribute(CONSTANT_VALUE);
        id.appendLineNumber(token.getLineNumber());

        if (sign != null) {
            errorHandler.flag(token, INVALID_CONSTANT, this);
        }

        return value;
    }
    else if (definition == null) {
        errorHandler.flag(token, NOT_CONSTANT_IDENTIFIER, this);
        return null;
    }
    else {
        errorHandler.flag(token, INVALID_CONSTANT, this);
        return null;
    }
}
```

[Listing 9-20](#) shows two versions of method `getConstantType()`. One version takes a constant's value as its argument, and it returns either the predefined `integer`, `real`, or `char` type specification, or it calls `TypeFactory.createStringType()` to return a string type specification. The other version takes a type identifier as its argument and looks up the identifier in the symbol table. If the identifier is a constant identifier or an enumeration constant identifier, the method returns the symbol table entry's type specification.

[Listing 9-20:](#) Method `getConstantType()` in class `ConstantDefinitionsParser`

```
/**
 * Return the type of a constant given its value.
 * @param value the constant value
 * @return the type specification.
 */
protected TypeSpec getConstantType(Object value)
{
    TypeSpec constantType = null;

    if (value instanceof Integer) {
        constantType = Predefined.integerType;
    }
    else if (value instanceof Float) {
        constantType = Predefined.realType;
    }
    else if (value instanceof String) {
        if (((String) value).length() == 1) {
            constantType = Predefined.charType;
        }
        else {
            constantType = TypeFactory.createStringType((String) value);
        }
    }

    return constantType;
}

/**
 * Return the type of a constant given its identifier.
 * @param identifier the constant&apos;s identifier.
 * @return the type specification.
 */
protected TypeSpec getConstantType(Token identifier)
{
    SymTabEntry id = symTabStack.lookup(identifier.getText());

    if (id == null) {
        return null;
    }

    Definition definition = id.getDefinition();

    if ((definition == CONSTANT) || (definition == ENUMERATION_CONSTANT)) {
        return id.getTypeSpec();
    }
    else {
        return null;
    }
}
```

Parsing Type Definitions and Type Specifications

Listing 9-21 shows the `parse()` method of the declarations parser subclass `TypeDefinitionsParser`.

Listing 9-21: Method `parse()` of class `TypeDefinitionsParser`

```java
// Synchronization set for a type identifier.
private static final EnumSet<PascalTokenType> IDENTIFIER_SET =
    DeclarationsParser.VAR_START_SET.clone();
static {
    IDENTIFIER_SET.add(IDENTIFIER);
}

// Synchronization set for the = token.
private static final EnumSet<PascalTokenType> EQUALS_SET =
    ConstantDefinitionsParser.CONSTANT_START_SET.clone();
static {
    EQUALS_SET.add(EQUALS);
    EQUALS_SET.add(SEMICOLON);
}

// Synchronization set for what follows a definition or declaration.
private static final EnumSet<PascalTokenType> FOLLOW_SET =
    EnumSet.of(SEMICOLON);

// Synchronization set for the start of the next definition or
// declaration.
private static final EnumSet<PascalTokenType> NEXT_START_SET =
    DeclarationsParser.VAR_START_SET.clone();
static {
    NEXT_START_SET.add(SEMICOLON);
    NEXT_START_SET.add(IDENTIFIER);
}

/**
 * Parse type definitions.
 * @param token the initial token.
 * @throws Exception if an error occurred.
 */
public void parse(Token token)
    throws Exception
{
    token = synchronize(IDENTIFIER_SET);

    // Loop to parse a sequence of type definitions
    // separated by semicolons.
    while (token.getType() == IDENTIFIER) {
        String name = token.getText().toLowerCase();
        SymTabEntry typeId = symTabStack.lookupLocal(name);

        // Enter the new identifier into the symbol table
        // but don&apos;t set how it&apos;s defined yet.
        if (typeId == null) {
            typeId = symTabStack.enterLocal(name);
            typeId.appendLineNumber(token.getLineNumber());
        }
        else {
            errorHandler.flag(token, IDENTIFIER_REDEFINED, this);
            typeId = null;
        }

        token = nextToken();  // consume the identifier token

        // Synchronize on the = token.
        token = synchronize(EQUALS_SET);
        if (token.getType() == EQUALS) {
            token = nextToken();  // consume the =
        }
        else {
            errorHandler.flag(token, MISSING_EQUALS, this);
        }

        // Parse the type specification.
        TypeSpecificationParser typeSpecificationParser =
            new TypeSpecificationParser(this);
        TypeSpec type = typeSpecificationParser.parse(token);

        // Set identifier to be a type and set its type specificationt.
        if (typeId != null) {
            typeId.setDefinition(TYPE);
        }

        // Cross-link the type identifier and the type specification.
        if ((type != null) && (typeId != null)) {
            if (type.getIdentifier() == null) {
                type.setIdentifier(typeId);
            }
            typeId.setTypeSpec(type);
        }
        else {
            token = synchronize(FOLLOW_SET);
        }

        token = currentToken();
        TokenType tokenType = token.getType();

        // Look for one or more semicolons after a definition.
        if (tokenType == SEMICOLON) {
            while (token.getType() == SEMICOLON) {
                token = nextToken();  // consume the ;
            }
        }

        // If at the start of the next definition or declaration,
        // then missing a semicolon.
        else if (NEXT_START_SET.contains(tokenType)) {
            errorHandler.flag(token, MISSING_SEMICOLON, this);
        }

        token = synchronize(IDENTIFIER_SET);
    }
}
```

Method `parse()` parses a sequence of type definitions separated by semicolons. It first searches the local symbol table to guard against redefining the type identifier within the same scope. It calls `typeSpecificationParser.parse()` to parse the type specification,

after which it sets the definition of the type identifier's symbol table entry to `DefinitionImpl.TYPE`. The method cross-links the type identifier's `SymTabEntry` object with the `TypeSpec` object.

Design Note

Delegate the parsing of all the Pascal type specifications to subclasses of `TypeSpecificationParser`. This is similar to how you delegated the parsing of all the Pascal statements to subclasses of `StatementParser`.

Listing 9-22 shows the `parse()` method of the Pascal parser subclass `TypeSpecificationParser`.

Listing 9-22: Method `parse()` of class `TypeSpecificationParser`

```
// Synchronization set for starting a type specification.
static final EnumSet<PascalTokenType> TYPE_START_SET =
    SimpleTypeParser.SIMPLE_TYPE_START_SET.clone();
static {
    TYPE_START_SET.add(PascalTokenType.ARRAY);
    TYPE_START_SET.add(PascalTokenType.RECORD);
    TYPE_START_SET.add(SEMICOLON);
}

/**
 * Parse a Pascal type specification.
 * @param token the current token.
 * @return the type specification.
 * @throws Exception if an error occurred.
 */
public TypeSpec parse(Token token)
    throws Exception
{
    // Synchronize at the start of a type specification.
    token = synchronize(TYPE_START_SET);

    switch ((PascalTokenType) token.getType()) {

        case ARRAY: {
            ArrayTypeParser arrayTypeParser = new ArrayTypeParser(this);
            return arrayTypeParser.parse(token);
        }

        case RECORD: {
            RecordTypeParser recordTypeParser = new RecordTypeParser(this);
            return recordTypeParser.parse(token);
        }

        default: {
            SimpleTypeParser simpleTypeParser = new SimpleTypeParser(this);
            return simpleTypeParser.parse(token);
        }
    }
}
```

The `parse()` method returns a `TypeSpec` object. If the current token is the keyword `ARRAY`, the method returns the result of `arrayTypeParser.parse()`. If it's the keyword `RECORD`, the method returns the result of `recordTypeParser.parse()`. Otherwise, the method returns the result of `simpleTypeParser.parse()`.

Simple Types

Listing 9-23 shows the `parse()` method of the type specification parser subclass `SimpleTypeParser`.

Listing 9-23: Method `parse()` of class `SimpleTypeParser`

```
// Synchronization set for starting a simple type specification.
static final EnumSet<PascalTokenType> SIMPLE_TYPE_START_SET =
    ConstantDefinitionsParser.CONSTANT_START_SET.clone();
static {
    SIMPLE_TYPE_START_SET.add(LEFT_PAREN);
    SIMPLE_TYPE_START_SET.add(COMMA);
    SIMPLE_TYPE_START_SET.add(SEMICOLON);
}

/**
 * Parse a simple Pascal type specification.
 * @param token the current token.
 * @return the simple type specification.
 * @throws Exception if an error occurred.
 */
public TypeSpec parse(Token token)
    throws Exception
{
    // Synchronize at the start of a simple type specification.
    token = synchronize(SIMPLE_TYPE_START_SET);

    switch ((PascalTokenType) token.getType()) {

        case IDENTIFIER: {
            String name = token.getText().toLowerCase();
            SymTabEntry id = symTabStack.lookup(name);

            if (id != null) {
                Definition definition = id.getDefinition();

                // It&apos;s either a type identifier
                // or the start of a subrange type.
                if (definition == DefinitionImpl.TYPE) {
                    id.appendLineNumber(token.getLineNumber());
                    token = nextToken();  // consume the identifier

                    // Return the type of the referent type.
                    return id.getTypeSpec();
                }
                else if ((definition != CONSTANT) &&
                         (definition != ENUMERATION_CONSTANT)) {
                    errorHandler.flag(token, NOT_TYPE_IDENTIFIER, this);
                    token = nextToken();  // consume the identifier
                    return null;
```

```
                }
                else {
                    SubrangeTypeParser subrangeTypeParser =
                        new SubrangeTypeParser(this);
                    return subrangeTypeParser.parse(token);
                }
            }
            else {
                errorHandler.flag(token, IDENTIFIER_UNDEFINED, this);
                token = nextToken();  // consume the identifier
                return null;
            }
        }

        case LEFT_PAREN: {
            EnumerationTypeParser enumerationTypeParser =
                new EnumerationTypeParser(this);
            return enumerationTypeParser.parse(token);
        }

        case COMMA:
        case SEMICOLON: {
            errorHandler.flag(token, INVALID_TYPE, this);
            return null;
        }

        default: {
            SubrangeTypeParser subrangeTypeParser =
                new SubrangeTypeParser(this);
            return subrangeTypeParser.parse(token);
        }
    }
}
```

If the current token is an identifier, the `parse()` method must determine whether it is a type identifier, a constant identifier, or an enumeration identifier. For a type identifier, the method returns the type specification of the identifier's symbol table entry. For a constant identifier or an enumeration identifier, the method returns the result of `subrangeTypeParser.parse()`.

If the current token is a left parenthesis, method `parse()` returns the result of `enumerationTypeParser.parse()`. For any other token, the method returns the result of `subrangeTypeParser.parse()`.

[Listing 9-24](#) shows the `parse()` and `checkValueType()` methods of the type specification parser subclass `SubrangeTypeParser`.

Listing 9-24: **Methods `parse()` and `checkValueType()` of class `SubrangeTypeParser`**

```
/**
 * Parse a Pascal subrange type specification.
 * @param token the current token.
 * @return the subrange type specification.
 * @throws Exception if an error occurred.
 */
public TypeSpec parse(Token token)
    throws Exception
{
    TypeSpec subrangeType = TypeFactory.createType(SUBRANGE);
    Object minValue = null;
    Object maxValue = null;

    // Parse the minimum constant.
    Token constantToken = token;
    ConstantDefinitionsParser constantParser =
        new ConstantDefinitionsParser(this);
    minValue = constantParser.parseConstant(token);

    // Set the minimum constant's type.
    TypeSpec minType = constantToken.getType() == IDENTIFIER
                        ? constantParser.getConstantType(constantToken)
                        : constantParser.getConstantType(minValue);

    minValue = checkValueType(constantToken, minValue, minType);

    token = currentToken();
    Boolean sawDotDot = false;

    // Look for the .. token.
    if (token.getType() == DOT_DOT) {
        token = nextToken();  // consume the .. token
        sawDotDot = true;
    }

    TokenType tokenType = token.getType();

    // At the start of the maximum constant?
    if (ConstantDefinitionsParser.CONSTANT_START_SET.contains(tokenType)) {
        if (!sawDotDot) {
            errorHandler.flag(token, MISSING_DOT_DOT, this);
        }

        // Parse the maximum constant.
        token = synchronize(ConstantDefinitionsParser.CONSTANT_START_SET);
        constantToken = token;
        maxValue = constantParser.parseConstant(token);

        // Set the maximum constant's type.
        TypeSpec maxType = constantToken.getType() == IDENTIFIER
                        ? constantParser.getConstantType(constantToken)
                        : constantParser.getConstantType(maxValue);

        maxValue = checkValueType(constantToken, maxValue, maxType);

        // Are the min and max value types valid?
        if ((minType == null) || (maxType == null)) {
            errorHandler.flag(constantToken, INCOMPATIBLE_TYPES, this);
        }

        // Are the min and max value types the same?
        else if (minType != maxType) {
            errorHandler.flag(constantToken, INVALID_SUBRANGE_TYPE, this);
        }

        // Min value > max value?
        else if ((minValue != null) && (maxValue != null) &&
                ((Integer) minValue >= (Integer) maxValue)) {
            errorHandler.flag(constantToken, MIN_GT_MAX, this);
        }
    }
    else {
        errorHandler.flag(constantToken, INVALID_SUBRANGE_TYPE, this);
```

```
    }

    subrangeType.setAttribute(SUBRANGE_BASE_TYPE, minType);
    subrangeType.setAttribute(SUBRANGE_MIN_VALUE, minValue);
    subrangeType.setAttribute(SUBRANGE_MAX_VALUE, maxValue);

    return subrangeType;
}

/**
 * Check a value of a type specification.
 * @param token the current token.
 * @param value the value.
 * @param type the type specifiction.
 * @return the value.
 */
private Object checkValueType(Token token, Object value, TypeSpec type)
{
    if (type == null) {
        return value;
    }
    if (type == Predefined.integerType) {
        return value;
    }
    else if (type == Predefined.charType) {
        char ch = ((String) value).charAt(0);
        return Character.getNumericValue(ch);
    }
    else if (type.getForm() == ENUMERATION) {
        return value;
    }
    else {
        errorHandler.flag(token, INVALID_SUBRANGE_TYPE, this);
        return value;
    }
}
```

The `parse()` method calls `TypeFactory.createType()` to create a subrange `TypeSpec` object. It calls `constantParser.parseConstant()` twice to parse the minimum and maximum constant values, which are separated by the `..` token. The method calls `checkValueType()` to verify that the values are type integer, character, or enumeration. It checks that the types of the two values are the same, and that the minimum value is less than or equal to the maximum value. If all is well, the method sets the subrange `TypeSpec` object's `SUBRANGE_BASE_TYPE`, `SUBRANGE_MIN_VALUE`, and `SUBRANGE_MAX_VALUE` attribute values.

Listing 9-25 shows the `parse()` and `parseEnumerationIdentifier()` methods of the type specification parser subclass `EnumerationTypeParser`.

Listing 9-25: Methods `parse()` and `parseEnumerationIdentifier()` of class `EnumerationTypeParser`

```
// Synchronization set to start an enumeration constant.
private static final EnumSet<PascalTokenType> ENUM_CONSTANT_START_SET =
    EnumSet.of(IDENTIFIER, COMMA);

// Synchronization set to follow an enumeration definition.
private static final EnumSet<PascalTokenType> ENUM_DEFINITION_FOLLOW_SET =
    EnumSet.of(RIGHT_PAREN, SEMICOLON);
static {
    ENUM_DEFINITION_FOLLOW_SET.addAll(DeclarationsParser.VAR_START_SET);
}

/**
 * Parse a Pascal enumeration type specification.
 * @param token the current token.
 * @return the enumeration type specification.
 * @throws Exception if an error occurred.
 */
public TypeSpec parse(Token token)
    throws Exception
{
    TypeSpec enumerationType = TypeFactory.createType(ENUMERATION);
    int value = -1;
    ArrayList<SymTabEntry> constants = new ArrayList<SymTabEntry>();

    token = nextToken();  // consume the opening (

    do {
        token = synchronize(ENUM_CONSTANT_START_SET);
        parseEnumerationIdentifier(token, ++value, enumerationType,
                                   constants);

        token = currentToken();
        TokenType tokenType = token.getType();

        // Look for the comma.
        if (tokenType == COMMA) {
            token = nextToken();  // consume the comma

            if (ENUM_DEFINITION_FOLLOW_SET.contains(token.getType())) {
                errorHandler.flag(token, MISSING_IDENTIFIER, this);
            }
        }
        else if (ENUM_CONSTANT_START_SET.contains(tokenType)) {
            errorHandler.flag(token, MISSING_COMMA, this);
        }
    } while (!ENUM_DEFINITION_FOLLOW_SET.contains(token.getType()));

    // Look for the closing ).
    if (token.getType() == RIGHT_PAREN) {
        token = nextToken();  // consume the )
    }
    else {
        errorHandler.flag(token, MISSING_RIGHT_PAREN, this);
    }

    enumerationType.setAttribute(ENUMERATION_CONSTANTS, constants);
    return enumerationType;
}

/**
 * Parse an enumeration identifier.
 * @param token the current token.
 * @param value the identifier&apos;s integer value (sequence number).
 * @param enumerationType the enumeration type specification.
 * @param constants the array of symbol table entries for the
 * enumeration constants.
 * @throws Exception if an error occurred.
 */
private void parseEnumerationIdentifier(Token token, int value,
```

```
                                          TypeSpec enumerationType,
                                          ArrayList<SymTabEntry> constants)
        throws Exception
    {
        TokenType tokenType = token.getType();

        if (tokenType == IDENTIFIER) {
            String name = token.getText().toLowerCase();
            SymTabEntry constantId = symTabStack.lookupLocal(name);

            if (constantId != null) {
                errorHandler.flag(token, IDENTIFIER_REDEFINED, this);
            }
            else {
                constantId = symTabStack.enterLocal(token.getText());
                constantId.setDefinition(ENUMERATION_CONSTANT);
                constantId.setTypeSpec(enumerationType);
                constantId.setAttribute(CONSTANT_VALUE, value);
                constantId.appendLineNumber(token.getLineNumber());
                constants.add(constantId);
            }

            token = nextToken();  // consume the identifier
        }
        else {
            errorHandler.flag(token, MISSING_IDENTIFIER, this);
        }
    }
```

The `parse()` method calls `TypeFactory.createType()` to create an enumeration `TypeSpec` object, and then it loops to call `parseEnumerationIdentifier()` for each enumeration constant identifier, incrementing the integer value by 1. It sets the enumeration `TypeSpec` object's `ENUMERATION_CONSTANTS` attribute value to the list of the symbol table entries of the enumeration constant identifiers. The integer value starts with 0 and increments by 1 for each successive enumeration constant.

Method `parseEnumerationIdentifier()` enters each enumeration constant identifier into the local symbol table. It sets the entry's definition to `DefinitionImpl.ENUMERATION_CONSTANT`, its type specification to the new enumeration `TypeSpec` object, and its `CONSTANT_VALUE` attribute value to the integer value. It appends the symbol table entry to the `constants` list.

Array Types

A Pascal array specification has two parts:

- After the reserved word `ARRAY`, a comma-separated list of one or more index type specifications, one per dimension, surrounded by square brackets, and
- After the reserved word `OF`, the element type specification.

Multidimensional array types are especially interesting. In Pascal, listing several index type specifications together, as in

```
ARRAY [1..3, 'a'..'z', boolean] OF real
```

is shorthand for

```
ARRAY [1..3] OF
    ARRAY ['a'..'z'] OF
        ARRAY [boolean] OF real;
```

Figure 9-6 shows the symbol table entries and type specification objects that are generated for these two equivalent type definitions. The figure shows how the array element types are linked together with the `ARRAY_ELEMENT_TYPE` attribute values of the array type specifications.

Listing 9-26 shows method `parse()` of the type specification parser subclass `ArrayTypeParser`.

Listing 9-26: Method **`parse()`** of class **`ArrayTypeParser`**

```
// Synchronization set for the [ token.
private static final EnumSet<PascalTokenType> LEFT_BRACKET_SET =
    SimpleTypeParser.SIMPLE_TYPE_START_SET.clone();
static {
    LEFT_BRACKET_SET.add(LEFT_BRACKET);
    LEFT_BRACKET_SET.add(RIGHT_BRACKET);
}

// Synchronization set for the ] token.
private static final EnumSet<PascalTokenType> RIGHT_BRACKET_SET =
    EnumSet.of(RIGHT_BRACKET, OF, SEMICOLON);

// Synchronization set for OF.
private static final EnumSet<PascalTokenType> OF_SET =
    TypeSpecificationParser.TYPE_START_SET.clone();
static {
    OF_SET.add(OF);
    OF_SET.add(SEMICOLON);
}

/**
 * Parse a Pascal array type specification.
 * @param token the current token.
 * @return the array type specification.
 * @throws Exception if an error occurred.
 */
public TypeSpec parse(Token token)
    throws Exception
{
    TypeSpec arrayType = TypeFactory.createType(ARRAY);
    token = nextToken();  // consume ARRAY

    // Synchronize at the [ token.
    token = synchronize(LEFT_BRACKET_SET);
    if (token.getType() != LEFT_BRACKET) {
        errorHandler.flag(token, MISSING_LEFT_BRACKET, this);
    }

    // Parse the list of index types.
    TypeSpec elementType = parseIndexTypeList(token, arrayType);

    // Synchronize at the ] token.
    token = synchronize(RIGHT_BRACKET_SET);
    if (token.getType() == RIGHT_BRACKET) {
        token = nextToken();  // consume [
    }
```

```
                else {
                    errorHandler.flag(token, MISSING_RIGHT_BRACKET, this);
                }

                // Synchronize at OF.
                token = synchronize(OF_SET);
                if (token.getType() == OF) {
                    token = nextToken();  // consume OF
                }
                else {
                    errorHandler.flag(token, MISSING_OF, this);
                }

                // Parse the element type.
                elementType.setAttribute(ARRAY_ELEMENT_TYPE, parseElementType(token));

                return arrayType;
            }
```
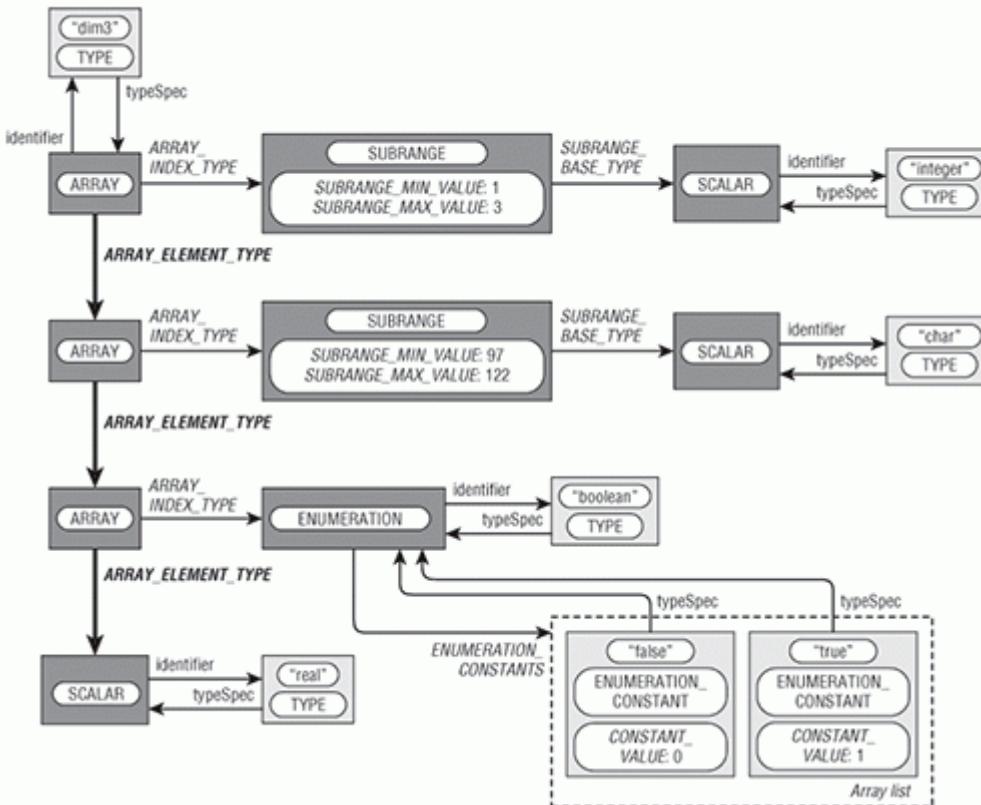
Figure 9-6: The two equivalent specifications for a 3-dimensional array generate the same configuration of symbol table entries and type specifications. Following the convention of Figure 9-3, SymTabEntry objects are in light gray and TypeSpec objects are in dark gray. The attribute value ARRAY_ELEMENT_TYPE is the link shown with bold arrows. In Pascal, the ordinal values of the characters 'a' and 'z' are 97 and 122, respectively.

```
dim3 = ARRAY [1..3, 'a'..'z', boolean] OF real;
dim3 = ARRAY [1..3] OF ARRAY ['a'..'z'] OF ARRAY [boolean] OF real;
```


Method parse() first calls TypeFactory.createType() to create an array TypeSpec object. It calls parseIndexTypeList() to parse the comma-separated list of one or more index type specifications. After consuming the OF reserved word, parse() calls parseElementType() to parse the element type specification.

Listing 9-27 shows methods parseIndexTypeList() and parseIndexType().

Listing 9-27: Methods `parseIndexTypeList()` and `parseIndexType()` of class `ArrayTypeParser`

```
        // Synchronization set to start an index type.
        private static final EnumSet<PascalTokenType> INDEX_START_SET =
            SimpleTypeParser.SIMPLE_TYPE_START_SET.clone();
        static {
            INDEX_START_SET.add(COMMA);
        }

        // Synchronization set to end an index type.
        private static final EnumSet<PascalTokenType> INDEX_END_SET =
            EnumSet.of(RIGHT_BRACKET, OF, SEMICOLON);

        // Synchronization set to follow an index type.
        private static final EnumSet<PascalTokenType> INDEX_FOLLOW_SET =
            INDEX_START_SET.clone();
        static {
            INDEX_FOLLOW_SET.addAll(INDEX_END_SET);
        }

        /**
         * Parse the list of index type specifications.
         * @param token the current token.
         * @param arrayType the current array type specification.
         * @return the element type specification.
         * @throws Exception if an error occurred.
         */
        private TypeSpec parseIndexTypeList(Token token, TypeSpec arrayType)
            throws Exception
        {
            TypeSpec elementType = arrayType;
            boolean anotherIndex = false;

            token = nextToken();  // consume the [ token
```

```java
            // Parse the list of index type specifications.
            do {
                anotherIndex = false;

                // Parse the index type.
                token = synchronize(INDEX_START_SET);
                parseIndexType(token, elementType);

                // Synchronize at the , token.
                token = synchronize(INDEX_FOLLOW_SET);
                TokenType tokenType = token.getType();
                if ((tokenType != COMMA) && (tokenType != RIGHT_BRACKET)) {
                    if (INDEX_START_SET.contains(tokenType)) {
                        errorHandler.flag(token, MISSING_COMMA, this);
                        anotherIndex = true;
                    }
                }

                // Create an ARRAY element type object
                // for each subsequent index type.
                else if (tokenType == COMMA) {
                    TypeSpec newElementType = TypeFactory.createType(ARRAY);
                    elementType.setAttribute(ARRAY_ELEMENT_TYPE, newElementType);
                    elementType = newElementType;

                    token = nextToken();  // consume the , token
                    anotherIndex = true;
                }
            } while (anotherIndex);

            return elementType;
        }

        /**
         * Parse an index type specification.
         * @param token the current token.
         * @param arrayType the current array type specification.
         * @throws Exception if an error occurred.
         */
        private void parseIndexType(Token token, TypeSpec arrayType)
            throws Exception
        {
            SimpleTypeParser simpleTypeParser = new SimpleTypeParser(this);
            TypeSpec indexType = simpleTypeParser.parse(token);
            arrayType.setAttribute(ARRAY_INDEX_TYPE, indexType);

            if (indexType == null) {
                return;
            }

            TypeForm form = indexType.getForm();
            int count = 0;

            // Check the index type and set the element count.
            if (form == SUBRANGE) {
                Integer minValue =
                    (Integer) indexType.getAttribute(SUBRANGE_MIN_VALUE);
                Integer maxValue =
                    (Integer) indexType.getAttribute(SUBRANGE_MAX_VALUE);

                if ((minValue != null) && (maxValue != null)) {
                    count = maxValue - minValue + 1;
                }
            }
            else if (form == ENUMERATION) {
                ArrayList<SymTabEntry> constants = (ArrayList<SymTabEntry>)
                    indexType.getAttribute(ENUMERATION_CONSTANTS);
                count = constants.size();
            }
            else {
                errorHandler.flag(token, INVALID_INDEX_TYPE, this);
            }
            arrayType.setAttribute(ARRAY_ELEMENT_COUNT, count);
        }
```

Method `parseIndexTypeList()` parses the list of index type specifications. It initializes variable `elementType` to the array `TypeSpec` object created by method `parse()`. In a loop, it calls `parseIndexType()` to parse each index type specification. After the first index type specification, `parseIndexTypeList()` must call `TypeFactory.createType()` to create an array `TypeSpec` object as the new element `TypeSpec` object. The method sets the `ARRAY_ELEMENT_TYPE` attribute value of the previous `elementType` to the new element `TypeSpec` object, and then it sets `elementType` to the new element `TypeSpec` object:

```java
TypeSpec newElementType = TypeFactory.createType(ARRAY);
elementType.setAttribute(ARRAY_ELEMENT_TYPE, newElementType);
elementType = newElementType;
```

This creates the `ARRAY_ELEMENT_TYPE` chain of array `TypeSpec` objects seen in Figure 9-6. The method returns the array `TypeSpec` object at the end of the chain.

Method `parseIndexType()` calls `simpleTypeParser.parse()` to parse each index type specification. It sets the `ARRAY_INDEX_TYPE` attribute value of the array `TypeSpec` object passed in by `parseIndexTypeList()`. The method checks a subrange type or an enumeration type. In either case, it gets the number of elements, which it sets as the `ARRAY_ELEMENT_COUNT` attribute value of the array `TypeSpec` object.

Finally, method `parseElementType()` returns the result of calling `typeSpecificationParser.parse()`, which parses the element type specification. See Listing 9-28.

Listing 9-28: Method `parseElementType()` of class `ArrayTypeParser`

```java
        /**
         * Parse the element type specification.
         * @param token the current token.
         * @return the element type specification.
         * @throws Exception if an error occurred.
         */
        private TypeSpec parseElementType(Token token)
            throws Exception
        {
            TypeSpecificationParser typeSpecificationParser =
                new TypeSpecificationParser(this);
            return typeSpecificationParser.parse(token);
        }
```

Record Types

The remaining Pascal type is the record type. Listing 9-29 shows the `parse()` method of the type specification parser subclass `RecordTypeParser`.

Listing 9-29: The `parse()` method of the parser subclass `RecordTypeParser`

```
// Synchronization set for the END.
private static final EnumSet<PascalTokenType> END_SET =
    DeclarationsParser.VAR_START_SET.clone();
static {
    END_SET.add(END);
    END_SET.add(SEMICOLON);
}

/**
 * Parse a Pascal record type specification.
 * @param token the current token.
 * @return the record type specification.
 * @throws Exception if an error occurred.
 */
public TypeSpec parse(Token token)
    throws Exception
{
    TypeSpec recordType = TypeFactory.createType(RECORD);
    token = nextToken();  // consume RECORD

    // Push a symbol table for the RECORD type specification.
    recordType.setAttribute(RECORD_SYMTAB, symTabStack.push());

    // Parse the field declarations.
    VariableDeclarationsParser variableDeclarationsParser =
        new VariableDeclarationsParser(this);
    variableDeclarationsParser.setDefinition(FIELD);
    variableDeclarationsParser.parse(token);

    // Pop off the record&apos;s symbol table.
    symTabStack.pop();

    // Synchronize at the END.
    token = synchronize(END_SET);

    // Look for the END.
    if (token.getType() == END) {
        token = nextToken();  // consume END
    }
    else {
        errorHandler.flag(token, MISSING_END, this);
    }

    return recordType;
}
```

As the syntax diagrams in Figure 9-1 show, field declarations inside of a record type definition and variable declarations have the same syntax. Therefore, the `parse()` method is relatively simple. It starts by calling `TypeFactory.createType()` to create a record `TypeSpec` object. Since a record type specification creates a new scope to contain its field identifiers, the statement

```
recordType.setAttribute(RECORD_SYMTAB, symTabStack.push());
```

pushes a new symbol table onto the symbol table stack and sets it as the record `TypeSpec` object's `RECORD_SYMTAB` attribute value. Then `parse()` sets the `definition` of `variableDeclarationsParser` to `FIELD` and calls `variableDeclarationsParser.parseFields()` to parse the field declarations. Afterwards, the method pops the record's symbol table off the stack. You'll examine class `VariableDeclarationsParser` next.

Parsing Variable Declarations

Now you're set to parse variable declarations. As mentioned above, the parser for variable declarations also parses record field declarations. Listings 9-16 and 9-29 showed that callers of this parser first set the `definition` field. Listing 9-30 shows the `parse()` method of the declarations parser subclass `VariableDeclarationsParser`.

Listing 9-30: Method `parse()` of class `VariableDeclarationsParser`

```
private Definition definition;  // how to define the identifier

// Synchronization set for a variable identifier.
static final EnumSet<PascalTokenType> IDENTIFIER_SET =
    DeclarationsParser.VAR_START_SET.clone();
static {
    IDENTIFIER_SET.add(IDENTIFIER);
    IDENTIFIER_SET.add(END);
    IDENTIFIER_SET.add(SEMICOLON);
}

// Synchronization set for the start of the next definition or declaration.
static final EnumSet<PascalTokenType> NEXT_START_SET =
    DeclarationsParser.ROUTINE_START_SET.clone();
static {
    NEXT_START_SET.add(IDENTIFIER);
    NEXT_START_SET.add(SEMICOLON);
}

/**
 * Parse variable declarations.
 * @param token the initial token.
 * @throws Exception if an error occurred.
 */
public void parse(Token token)
    throws Exception
{
    token = synchronize(IDENTIFIER_SET);

    // Loop to parse a sequence of variable declarations
    // separated by semicolons.
```

```
        while (token.getType() == IDENTIFIER) {

            // Parse the identifier sublist and its type specification.
            parseIdentifierSublist(token);

            token = currentToken();
            TokenType tokenType = token.getType();

            // Look for one or more semicolons after a definition.
            if (tokenType == SEMICOLON) {
                while (token.getType() == SEMICOLON) {
                    token = nextToken();  // consume the ;
                }
            }

            // If at the start of the next definition or declaration,
            // then missing a semicolon.
            else if (NEXT_START_SET.contains(tokenType)) {
                errorHandler.flag(token, MISSING_SEMICOLON, this);
            }

            token = synchronize(IDENTIFIER_SET);
        }
    }
```

Method `parse()` loops to parse the variable declarations separated by semicolons. It calls `parseIdentifierSublist()` to parse each comma-separated sublist of one or more variable identifiers and their type specification. See Listing 9-31.

Listing 9-31: Method `parseIdentifierSublist()` of class `VariableDeclarationsParser`

```java
// Synchronization set to start a sublist identifier.
static final EnumSet<PascalTokenType> IDENTIFIER_START_SET =
    EnumSet.of(IDENTIFIER, COMMA);

// Synchronization set to follow a sublist identifier.
private static final EnumSet<PascalTokenType> IDENTIFIER_FOLLOW_SET =
    EnumSet.of(COLON, SEMICOLON);
static {
    IDENTIFIER_FOLLOW_SET.addAll(DeclarationsParser.VAR_START_SET);
}

// Synchronization set for the , token.
private static final EnumSet<PascalTokenType> COMMA_SET =
    EnumSet.of(COMMA, COLON, IDENTIFIER, SEMICOLON);

/**
 * Parse a sublist of identifiers and their type specification.
 * @param token the current token.
 * @return the sublist of identifiers in a declaration.
 * @throws Exception if an error occurred.
 */
protected ArrayList<SymTabEntry> parseIdentifierSublist(Token token)
    throws Exception
{
    ArrayList<SymTabEntry> sublist = new ArrayList<SymTabEntry>();

    do {
        token = synchronize(IDENTIFIER_START_SET);
        SymTabEntry id = parseIdentifier(token);

        if (id != null) {
            sublist.add(id);
        }

        token = synchronize(COMMA_SET);
        TokenType tokenType = token.getType();

        // Look for the comma.
        if (tokenType == COMMA) {
            token = nextToken();  // consume the comma

            if (IDENTIFIER_FOLLOW_SET.contains(token.getType())) {
                errorHandler.flag(token, MISSING_IDENTIFIER, this);
            }
        }
        else if (IDENTIFIER_START_SET.contains(tokenType)) {
            errorHandler.flag(token, MISSING_COMMA, this);
        }
    } while (!IDENTIFIER_FOLLOW_SET.contains(token.getType()));

    // Parse the type specification.
    TypeSpec type = parseTypeSpec(token);

    // Assign the type specification to each identifier in the list.
    for (SymTabEntry variableId : sublist) {
        variableId.setTypeSpec(type);
    }

    return sublist;
}
```

Method `parseIdentifierSublist()` loops over the comma-separated sublist of variable identifiers before the : token and calls `parseIdentifier()` to parse each identifier. It then calls `parseTypeSpec()` to parse the type spcification. The `for` loop sets the type specification of the symbol table entry of each identifier in the sublist. Listing 9-32 shows methods `parseIdentifier()` and `parseTypeSpec()`.

Listing 9-32: Methods `parseIdentifier()` and `parseTypeSpec()` of class `VariableDeclarationsParser`

```java
/**
 * Parse an identifier.
 * @param token the current token.
 * @return the symbol table entry of the identifier.
 * @throws Exception if an error occurred.
 */
private SymTabEntry parseIdentifier(Token token)
    throws Exception
{
    SymTabEntry id = null;

    if (token.getType() == IDENTIFIER) {
        String name = token.getText().toLowerCase();
        id = symTabStack.lookupLocal(name);
```

```
                // Enter a new identifier into the symbol table.
                if (id == null) {
                    id = symTabStack.enterLocal(name);
                    id.setDefinition(definition);
                    id.appendLineNumber(token.getLineNumber());
                }
                else {
                    errorHandler.flag(token, IDENTIFIER_REDEFINED, this);
                }

                token = nextToken();   // consume the identifier token
            }
            else {
                errorHandler.flag(token, MISSING_IDENTIFIER, this);
            }

            return id;
        }

        // Synchronization set for the : token.
        private static final EnumSet<PascalTokenType> COLON_SET =
            EnumSet.of(COLON, SEMICOLON);

        /**
         * Parse the type specification.
         * @param token the current token.
         * @return the type specification.
         * @throws Exception if an error occurs.
         */
        protected TypeSpec parseTypeSpec(Token token)
            throws Exception
        {
            // Synchronize on the : token.
            token = synchronize(COLON_SET);
            if (token.getType() == COLON) {
                token = nextToken(); // consume the :
            }
            else {
                errorHandler.flag(token, MISSING_COLON, this);
            }

            // Parse the type specification.
            TypeSpecificationParser typeSpecificationParser =
                new TypeSpecificationParser(this);
            TypeSpec type = typeSpecificationParser.parse(token);

            return type;
        }
```

Method `parseIdentifier()` parses each variable identifier and enters it into the symbol table. It checks for a redefinition of the identifier within the current scope. It sets the entry's definition and appends the current line number to the entry.

Method `parseTypeSpec()` calls `typeSpecificationParser.parse()` to parse and return the type specification.

Design Note

You've removed the first three Chapter 6 Hacks. Each variable in the source program now has a data type, which eliminates Hack #1. You fixed Hack #2 when you entered each variable into the local symbol table when it was declared. Hack #3 no longer applies now that you can have more than one symbol table on the stack.

Program 9: Pascal Cross-Referencer II

Now expand the cross-referencer and the parse tree printer that you first developed in Chapters 4 and 5, respectively. But first, modify the constructor of the main class `Pascal`, as shown in

. The intermediate code variable `iCode` now is set from the `ROUTINE_ICODE` attribute value of the program identifier's symbol table entry.

Listing 9-33: Modifications to the constructor of the main class `Pascal`

```
        if (parser.getErrorCount() == 0) {
            symTabStack = parser.getSymTabStack();

            SymTabEntry programId = symTabStack.getProgramId();
            iCode = (ICode) programId.getAttribute(ROUTINE_ICODE);

            if (xref) {
                CrossReferencer crossReferencer = new CrossReferencer();
                crossReferencer.print(symTabStack);
            }

            if (intermediate) {
                ParseTreePrinter treePrinter =
                                    new ParseTreePrinter(System.out);
                treePrinter.print(symTabStack);
            }

            backend.process(iCode, symTabStack);
        }
```

Design Note

The cross-reference utility is not only useful in its own right, but in its complete version, it performs critical verification of the contents of the symbol tables for the main program, each record type definition, and each procedure and function.

> Code and data structures as complex as the symbol tables that you've developed in this chapter require extraordinary means such as the cross-reference utility to ensure that all the data and references are correct. Then you can be confident that the back end executors and code generators will function properly when they rely upon these symbol tables.

Listing 9-34 shows the new versions of methods `print()`, `printColumnHeadings()`, and `printSymTab()` and new method `printRoutine()` in the utility class `CrossReferencer`.

Listing 9-34: Methods `print()`, `printColumnHeadings()`, `printSymTab()`, and `printRoutine()` in class `CrossReferencer`

```
/**
 * Print the cross-reference table.
 * @param symTabStack the symbol table stack.
 */
public void print(SymTabStack symTabStack)
{
    System.out.println("\n===== CROSS-REFERENCE TABLE =====");

    SymTabEntry programId = symTabStack.getProgramId();
    printRoutine(programId);
}

/**
 * Print a cross-reference table for a routine.
 * @param routineId the routine identifier&apos;s symbol table entry.
 */
private void printRoutine(SymTabEntry routineId)
{
    Definition definition = routineId.getDefinition();
    System.out.println("\n*** " + definition.toString() +
                       " " + routineId.getName() + " ***");
    printColumnHeadings();

    // Print the entries in the routine&apos;s symbol table.
    SymTab symTab = (SymTab) routineId.getAttribute(ROUTINE_SYMTAB);
    ArrayList<TypeSpec> newRecordTypes = new ArrayList<TypeSpec>();
    printSymTab(symTab, newRecordTypes);

    // Print cross-reference tables for any records defined in the routine.
    if (newRecordTypes.size() > 0) {
        printRecords(newRecordTypes);
    }

    // Print any procedures and functions defined in the routine.
    ArrayList<SymTabEntry> routineIds =
        (ArrayList<SymTabEntry>) routineId.getAttribute(ROUTINE_ROUTINES);
    if (routineIds != null) {
        for (SymTabEntry rtnId : routineIds) {
            printRoutine(rtnId);
        }
    }
}

/**
 * Print column headings.
 */
private void printColumnHeadings()
{
    System.out.println();
    System.out.println(String.format(NAME_FORMAT, "Identifier")
                       + NUMBERS_LABEL +    "Type specification");
    System.out.println(String.format(NAME_FORMAT, "----------")
                       + NUMBERS_UNDERLINE + "------------------");
}

/**
 * Print the entries in a symbol table.
 * @param symTab the symbol table.
 * @param recordTypes the list to fill with RECORD type specifications.
 */
private void printSymTab(SymTab symTab, ArrayList<TypeSpec> recordTypes)
{
    // Loop over the sorted list of symbol table entries.
    ArrayList<SymTabEntry> sorted = symTab.sortedEntries();
    for (SymTabEntry entry : sorted) {
        ArrayList<Integer> lineNumbers = entry.getLineNumbers();

        // For each entry, print the identifier name
        // followed by the line numbers.
        System.out.print(String.format(NAME_FORMAT, entry.getName()));
        if (lineNumbers != null) {
            for (Integer lineNumber : lineNumbers) {
                System.out.print(String.format(NUMBER_FORMAT, lineNumber));
            }
        }

        // Print the symbol table entry.
        System.out.println();
        printEntry(entry, recordTypes);
    }
}
```

Method `print()` calls `symTabStack.getProgramId()` to get the symbol table entry of the Pascal program identifier and then calls `printRoutine()`.

Method `printRoutine()` gets the `ROUTINE_SYMTAB` attribute value of the routine identifier's symbol table entry. This value is the symbol table that holds all the identifiers defined within the routine. The method calls `printSymTab()` to print the symbol table contents and to build an array list of any record types defined by the routine. Method `printRecords()` prints the details of these record types. At the end, method `printRoutine()` calls itself recursively to handle any procedures and functions defined in the routine.

Besides simply printing each symbol table entry's identifier name and line numbers, method `printSymTab()` now must also call `printEntry()` to print the contents of each entry.

Listing 9-35 shows methods `printEntry()` and `toString()`.

Listing 9-35: Methods `printEntry()` and `toString()` of class `CrossReferencer`

```
/**
 * Print a symbol table entry.
 * @param entry the symbol table entry.
 * @param recordTypes the list to fill with RECORD type specifications.
```

```java
     */
    private void printEntry(SymTabEntry entry, ArrayList<TypeSpec> recordTypes)
    {
        Definition definition = entry.getDefinition();
        int nestingLevel = entry.getSymTab().getNestingLevel();
        System.out.println(INDENT + "Defined as: " + definition.getText());
        System.out.println(INDENT + "Scope nesting level: " + nestingLevel);

        // Print the type specification.
        TypeSpec type = entry.getTypeSpec();
        printType(type);

        switch ((DefinitionImpl) definition) {

            case CONSTANT: {
                Object value = entry.getAttribute(CONSTANT_VALUE);
                System.out.println(INDENT + "Value = " + toString(value));

                // Print the type details only if the type is unnamed.
                if (type.getIdentifier() == null) {
                    printTypeDetail(type, recordTypes);
                }

                break;
            }

            case ENUMERATION_CONSTANT: {
                Object value = entry.getAttribute(CONSTANT_VALUE);
                System.out.println(INDENT + "Value = " + toString(value));

                break;
            }

            case TYPE: {

                // Print the type details only when the type is first defined.
                if (entry == type.getIdentifier()) {
                    printTypeDetail(type, recordTypes);
                }

                break;
            }

            case VARIABLE: {

                // Print the type details only if the type is unnamed.
                if (type.getIdentifier() == null) {
                    printTypeDetail(type, recordTypes);
                }

                break;
            }
        }
    }

    /**
     * Convert a value to a string.
     * @param value the value.
     * @return the string.
     */
    private String toString(Object value)
    {
        return value instanceof String ? "&apos;" + (String) value + "&apos;"
                                       : value.toString();
    }
```

Method `printEntry()` calls `printType()` to print the identifier's type specification. Then it prints information from the symbol table entry that is pertinent to the way the identifier is defined. For a constant identifier, it prints the constant value. For a constant identifier or an enumeration constant, it prints the constant's value. It prints the type details of a type identifier by calling `printTypeDetail()` the first time the type is defined. For a variable identifier, it calls `printTypeDetail()` only if the variable's type is unnamed.

shows methods `printType()` and `printTypeDetail()`.

Listing 9-36: Methods `printType()` and `printTypeDetail()` of class `CrossReferencer`

```java
    /**
     * Print a type specification.
     * @param type the type specification.
     */
    private void printType(TypeSpec type)
    {
        if (type != null) {
            TypeForm form = type.getForm();
            SymTabEntry typeId = type.getIdentifier();
            String typeName = typeId != null ? typeId.getName() : "<unnamed>";

            System.out.println(INDENT + "Type form = " + form +
                               ", Type id = " + typeName);
        }
    }

    private static final String ENUM_CONST_FORMAT = "%" + NAME_WIDTH + "s = %s";

    /**
     * Print the details of a type specification.
     * @param type the type specification.
     * @param recordTypes the list to fill with RECORD type specifications.
     */
    private void printTypeDetail(TypeSpec type, ArrayList<TypeSpec> recordTypes)
    {
        TypeForm form = type.getForm();

        switch ((TypeFormImpl) form) {

            case ENUMERATION: {
                ArrayList<SymTabEntry> constantIds = (ArrayList<SymTabEntry>)
                    type.getAttribute(ENUMERATION_CONSTANTS);

                System.out.println(INDENT + "--- Enumeration constants ---");

                // Print each enumeration constant and its value.
                for (SymTabEntry constantId : constantIds) {
                    String name = constantId.getName();
                    Object value = constantId.getAttribute(CONSTANT_VALUE);

                    System.out.println(INDENT + String.format(ENUM_CONST_FORMAT,
```

```
                                          name, value));
            }

            break;
        }

        case SUBRANGE: {
            Object minValue = type.getAttribute(SUBRANGE_MIN_VALUE);
            Object maxValue = type.getAttribute(SUBRANGE_MAX_VALUE);
            TypeSpec baseTypeSpec =
                (TypeSpec) type.getAttribute(SUBRANGE_BASE_TYPE);

            System.out.println(INDENT + "--- Base type ---");
            printType(baseTypeSpec);

            // Print the base type details only if the type is unnamed.
            if (baseTypeSpec.getIdentifier() == null) {
                printTypeDetail(baseTypeSpec, recordTypes);
            }

            System.out.print(INDENT + "Range = ");
            System.out.println(toString(minValue) + ".." +
                               toString(maxValue));

            break;
        }

        case ARRAY: {
            TypeSpec indexType =
                (TypeSpec) type.getAttribute(ARRAY_INDEX_TYPE);
            TypeSpec elementType =
                (TypeSpec) type.getAttribute(ARRAY_ELEMENT_TYPE);
            int count = (Integer) type.getAttribute(ARRAY_ELEMENT_COUNT);

            System.out.println(INDENT + "--- INDEX TYPE ---");
            printType(indexType);

            // Print the index type details only if the type is unnamed.
            if (indexType.getIdentifier() == null) {
                printTypeDetail(indexType, recordTypes);
            }

            System.out.println(INDENT + "--- ELEMENT TYPE ---");
            printType(elementType);
            System.out.println(INDENT.toString() + count + " elements");

            // Print the element type details only if the type is unnamed.
            if (elementType.getIdentifier() == null) {
                printTypeDetail(elementType, recordTypes);
            }

            break;
        }

        case RECORD: {
            recordTypes.add(type);
            break;
        }
    }
}
```

What detailed type information method `printTypeDetail()` prints depends on the type form. For an enumeration type, it prints the enumeration constants and their values. For a subrange type, it prints information about the base type and the minimum and maximum values. For an array type, it prints information about the index type and the element type. Finally, for a record type, the method simply adds the record type specification to the array list of record types.

For each record type specification in the array list, method `printRecords()` gets the `RECORD_SYMTAB` attribute value, which is the symbol table for the record type's fields. The method calls `printSymTab()` to print the cross-reference listing of the fields. It also calls `printRecords()` to print any nested record types. See Listing 9-37.

Listing 9-37: Method `printRecords()` of class `CrossReferencer`

```
/**
 * Print cross-reference tables for records defined in the routine.
 * @param recordTypes the list to fill with RECORD type specifications.
 */
private void printRecords(ArrayList<TypeSpec> recordTypes)
{
    for (TypeSpec recordType : recordTypes) {
        SymTabEntry recordId = recordType.getIdentifier();
        String name = recordId != null ? recordId.getName() : "<unnamed>";

        System.out.println("\n--- RECORD " + name + " ---");
        printColumnHeadings();

        // Print the entries in the record&apos;s symbol table.
        SymTab symTab = (SymTab) recordType.getAttribute(RECORD_SYMTAB);
        ArrayList<TypeSpec> newRecordTypes = new ArrayList<TypeSpec>();
        printSymTab(symTab, newRecordTypes);

        // Print cross-reference tables for any nested records.
        if (newRecordTypes.size() > 0) {
            printRecords(newRecordTypes);
        }
    }
}
```

So methods `printEntry()`, `printType()`, and `printTypeDetail()` together generate the printed cross-reference and type information for each identifier. The following are example printouts for various types of identifiers:

- Constant identifier

```
epsilon          003 005
                              Defined as: constant
                              Scope nesting level: 1
                              Type form = scalar, Type id = real
                              value = 1.0E-6
```

- Enumeration constant

```
friday           017 018
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = week
                              value = 4
```

- **Variable with a named type**

```
root          067
                        Defined as: variable
                        Scope nesting level: 1
                        Type form = scalar, Type id = real
```

- **Variable with an unnamed type**

```
var8          046
                        Defined as: variable
                        Scope nesting level: 1
                        Type form = enumeration, Type id = <unnamed>
                        --- Enumeration constants ---
                                    fee = 0
                                    fye = 1
                                    foe = 2
                                    fum = 3
```

- **Enumeration type**

```
week          017
                        Defined as: type
                        Scope nesting level: 1
                        Type form = enumeration, Type id = week
                        --- Enumeration constants ---
                               monday = 0
                              tuesday = 1
                            wednesday = 2
                             thursday = 3
                               friday = 4
                             saturday = 5
                               sunday = 6
```

- **Subrange type**

```
weekday       018
                        Defined as: type
                        Scope nesting level: 1
                        Type form = subrange, Type id = weekend
                        --- Base type ---
                        Type form = enumeration, Type id = week
                        Range = 0..4
```

- **Array type (single dimension, unnamed base type, named element type)**

```
ar2           022
                        Defined as: type
                        Scope nesting level: 1
                        Type form = array, Type id = ar2
                        --- INDEX TYPE ---
                        Type form = enumeration, Type id = <unnamed>
                        --- Enumeration constants ---
                                  alpha = 0
                                   beta = 1
                                  gamma = 2
                        --- ELEMENT TYPE ---
                        Type form = subrange, Type id = range2
                        3 elements
```

- **Array type (3-dimensional)**

```
ar5           025 063
                        Defined as: type
                        Scope nesting level: 1
                        Type form = array, Type id = ar5
                        --- INDEX TYPE ---
                        Type form = subrange, Type id = range1
                        --- ELEMENT TYPE ---
                        Type form = array, Type id = <unnamed>
                        11 elements
                        --- INDEX TYPE ---
                        Type form = subrange, Type id = range2
                        --- ELEMENT TYPE ---
                        Type form = array, Type id = <unnamed>
                        17 elements
                        --- INDEX TYPE ---
                        Type form = subrange, Type id = <unnamed>
                        --- Base type ---
                        Type form = enumeration, Type id = enum1
                        Range = 2..4
                        --- ELEMENT TYPE ---
                        Type form = enumeration, Type id = enum1
                        3 elements
```

Listing 9-38 shows output generated from a sample source file `declarations.txt`. The command line is similar to

```
java -classpath classes Pascal execute -x declarations.txt
```

Because you haven't yet modified any of the statement parsers, the source file has only an "empty" compound statement.

Listing 9-38: Cross-reference output

```
001 CONST
002     ten = 10;
003     epsilon = 1.0E-6;
004     x = &apos;x&apos;;
005     limit = -epsilon;
006     hello = &apos;Hello, world!&apos;;
007
008 TYPE
009     range1 = 0..ten;
010     range2 = &apos;a&apos;..&apos;q&apos;;
011     range3 = range1;
012
013     enum1 = (a, b, c, d, e);
014     enum2 = enum1;
015     range4 = b..d;
016
017     week = (monday, tuesday, wednesday, thursday, friday, saturday, sunday);
018     weekday = monday..friday;
019     weekend = saturday..sunday;
020
021     ar1 = ARRAY [range1] OF integer;
022     ar2 = ARRAY [(alpha, beta, gamma)] OF range2;
023     ar3 = ARRAY [enum2] OF ar1;
024     ar4 = ARRAY [range3] OF (foo, bar, baz);
```

```
025     ar5 = ARRAY [range1] OF ARRAY[range2] OF ARRAY[c..e] OF enum2;
026     ar6 = ARRAY [range1, range2, c..e] OF enum2;
027
028     rec1 = RECORD
029               i : integer;
030               r : real;
031               b1, b2 : boolean;
032               c : char
033            END;
034
035     ar7 = ARRAY [range2] OF RECORD
036                             ten : integer;
037                             r : rec1;
038                             a : ARRAY[range4] OF range2;
039                          END;
040
041 VAR
042     var1 : integer;
043     var2, var3 : range2;
044     var4 : enum2;
045     var5, var6, var7 : -7..ten;
046     var8 : (fee, fye, foe, fum);
047     var9 : range3;
048
049     var10 : rec1;
050     var11 : RECORD
051               b : boolean;
052               r : RECORD
053                     aa : ar1;
054                     bb : boolean;
055                     r  : real;
056                     v1 : ar6;
057                     v2 : ARRAY [enum1, range1] OF ar7;
058                  END;
059               a : ARRAY [1..5] OF boolean;
060            END;
061
062     var12 : ar1;
063     var15 : ar5;
064     var16 : ar6;
065
066     number : range1;
067     root : real;
068
069 BEGIN
070 END.


                70 source lines.

                 0 syntax errors.
               0.11 seconds total parsing time.


===== CROSS-REFERENCE TABLE =====

*** PROGRAM dummyprogramname ***


Identifier      Line numbers    Type specification
----------      ------------    ------------------
a               013
                                Defined as: enumeration constant
                                Scope nesting level: 1
                                Type form = enumeration, Type id = enum1
                                Value = 0
alpha           022
                                Defined as: enumeration constant
                                Scope nesting level: 1
                                Type form = enumeration, Type id = <unnamed>
                                Value = 0
ar1             021 023 053 062
                                Defined as: type
                                Scope nesting level: 1
                                Type form = array, Type id = ar1
                                --- INDEX TYPE ---
                                Type form = subrange, Type id = range1
                                --- ELEMENT TYPE ---
                                Type form = scalar, Type id = integer
                                11 elements
ar2             022
                                Defined as: type
                                Scope nesting level: 1
                                Type form = array, Type id = ar2
                                --- INDEX TYPE ---
                                Type form = enumeration, Type id = <unnamed>
                                --- Enumeration constants ---
                                        alpha = 0
                                         beta = 1
                                        gamma = 2
                                --- ELEMENT TYPE ---
                                Type form = subrange, Type id = range2
                                3 elements
ar3             023
                                Defined as: type
                                Scope nesting level: 1
                                Type form = array, Type id = ar3
                                --- INDEX TYPE ---
                                Type form = enumeration, Type id = enum1
                                --- ELEMENT TYPE ---
                                Type form = array, Type id = ar1
                                5 elements
ar4             024
                                Defined as: type
                                Scope nesting level: 1
                                Type form = array, Type id = ar4
                                --- INDEX TYPE ---
                                Type form = subrange, Type id = range1
                                --- ELEMENT TYPE ---
                                Type form = enumeration, Type id = <unnamed>
                                11 elements
                                --- Enumeration constants ---
                                        foo = 0
                                        bar = 1
                                        baz = 2
ar5             025 063
                                Defined as: type
                                Scope nesting level: 1
                                Type form = array, Type id = ar5
                                --- INDEX TYPE ---
                                Type form = subrange, Type id = range1
                                --- ELEMENT TYPE ---
                                Type form = array, Type id = <unnamed>
                                11 elements
                                --- INDEX TYPE ---
                                Type form = subrange, Type id = range2
```

```
                              --- ELEMENT TYPE ---
                              Type form = array, Type id = <unnamed>
                              17 elements
                              --- INDEX TYPE ---
                              Type form = subrange, Type id = <unnamed>
                              --- Base type ---
                              Type form = enumeration, Type id = enum1
                              Range = 2..4
                              --- ELEMENT TYPE ---
                              Type form = enumeration, Type id = enum1
                              3 elements
ar6            026 056 064
                              Defined as: type
                              Scope nesting level: 1
                              Type form = array, Type id = ar6
                              --- INDEX TYPE ---
                              Type form = subrange, Type id = range1
                              --- ELEMENT TYPE ---
                              Type form = array, Type id = <unnamed>
                              11 elements
                              --- INDEX TYPE ---
                              Type form = subrange, Type id = range2
                              --- ELEMENT TYPE ---
                              Type form = array, Type id = <unnamed>
                              17 elements
                              --- INDEX TYPE ---
                              Type form = subrange, Type id = <unnamed>
                              --- Base type ---
                              Type form = enumeration, Type id = enum1
                              Range = 2..4
                              --- ELEMENT TYPE ---
                              Type form = enumeration, Type id = enum1
                              3 elements
ar7            035 057
                              Defined as: type
                              Scope nesting level: 1
                              Type form = array, Type id = ar7
                              --- INDEX TYPE ---
                              Type form = subrange, Type id = range2
                              --- ELEMENT TYPE ---
                              Type form = record, Type id = <unnamed>
                              17 elements
b              013 015
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum1
                              Value = 1
bar            024
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = <unnamed>
                              Value = 1
baz            024
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = <unnamed>
                              Value = 2
beta           022
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = <unnamed>
                              Value = 1
c              013 025 026
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum1
                              Value = 2
d              013 015
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum1
                              Value = 3
e              013 025 026
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum1
                              Value = 4
enum1          013 014 057
                              Defined as: type
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum1
                              --- Enumeration constants ---
                                        a = 0
                                        b = 1
                                        c = 2
                                        d = 3
                                        e = 4
enum2          014 023 025 026 044
                              Defined as: type
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum1
epsilon        003 005
                              Defined as: constant
                              Scope nesting level: 1
                              Type form = scalar, Type id = real
                              Value = 1.0E-6
fee            046
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = <unnamed>
                              Value = 0
foe            046
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = <unnamed>
                              Value = 2
foo            024
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = <unnamed>
                              Value = 0
friday         017 018
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = week
                              Value = 4
fum            046
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = <unnamed>
                              Value = 3
fye            046
```

```
                                   Defined as: enumeration constant
                                   Scope nesting level: 1
                                   Type form = enumeration, Type id = <unnamed>
                                   Value = 1
gamma          022
                                   Defined as: enumeration constant
                                   Scope nesting level: 1
                                   Type form = enumeration, Type id = <unnamed>
                                   Value = 2
hello          006
                                   Defined as: constant
                                   Scope nesting level: 1
                                   Type form = array, Type id = <unnamed>
                                   Value = &apos;Hello, world!&apos;
                                   --- INDEX TYPE ---
                                   Type form = subrange, Type id = <unnamed>
                                   --- Base type ---
                                   Type form = scalar, Type id = integer
                                   Range = 1..13
                                   --- ELEMENT TYPE ---
                                   Type form = scalar, Type id = char
                                   13 elements
limit          005
                                   Defined as: constant
                                   Scope nesting level: 1
                                   Type form = scalar, Type id = real
                                   Value = -1.0E-6
monday         017 018
                                   Defined as: enumeration constant
                                   Scope nesting level: 1
                                   Type form = enumeration, Type id = week
                                   Value = 0
number         066
                                   Defined as: variable
                                   Scope nesting level: 1
                                   Type form = subrange, Type id = range1
range1         009 011 021 025 026 057 066
                                   Defined as: type
                                   Scope nesting level: 1
                                   Type form = subrange, Type id = range1
                                   --- Base type ---
                                   Type form = scalar, Type id = integer
                                   Range = 0..10
range2         010 022 025 026 035 038 043
                                   Defined as: type
                                   Scope nesting level: 1
                                   Type form = subrange, Type id = range2
                                   --- Base type ---

                                   Type form = scalar, Type id = char
                                   Range = 10..26
range3         011 024 047
                                   Defined as: type
                                   Scope nesting level: 1
                                   Type form = subrange, Type id = range1
range4         015 038
                                   Defined as: type
                                   Scope nesting level: 1
                                   Type form = subrange, Type id = range4
                                   --- Base type ---
                                   Type form = enumeration, Type id = enum1
                                   Range = 1..3
rec1           028 037 049
                                   Defined as: type
                                   Scope nesting level: 1
                                   Type form = record, Type id = rec1
root           067
                                   Defined as: variable
                                   Scope nesting level: 1
                                   Type form = scalar, Type id = real
saturday       017 019
                                   Defined as: enumeration constant
                                   Scope nesting level: 1
                                   Type form = enumeration, Type id = week
                                   Value = 5
sunday         017 019
                                   Defined as: enumeration constant
                                   Scope nesting level: 1
                                   Type form = enumeration, Type id = week
                                   Value = 6
ten            002 009 045
                                   Defined as: constant
                                   Scope nesting level: 1
                                   Type form = scalar, Type id = integer
                                   Value = 10
thursday       017
                                   Defined as: enumeration constant
                                   Scope nesting level: 1
                                   Type form = enumeration, Type id = week
                                   Value = 3
tuesday        017
                                   Defined as: enumeration constant
                                   Scope nesting level: 1
                                   Type form = enumeration, Type id = week
                                   Value = 1
var1           042
                                   Defined as: variable
                                   Scope nesting level: 1
                                   Type form = scalar, Type id = integer
var10          049
                                   Defined as: variable
                                   Scope nesting level: 1
                                   Type form = record, Type id = rec1
var11          050
                                   Defined as: variable
                                   Scope nesting level: 1
                                   Type form = record, Type id = <unnamed>
var12          062
                                   Defined as: variable
                                   Scope nesting level: 1
                                   Type form = array, Type id = ar1
var15          063
                                   Defined as: variable
                                   Scope nesting level: 1
                                   Type form = array, Type id = ar5
var16          064
                                   Defined as: variable
                                   Scope nesting level: 1
                                   Type form = array, Type id = ar6
var2           043
                                   Defined as: variable
                                   Scope nesting level: 1
                                   Type form = subrange, Type id = range2
```

```
var3            043
                            Defined as: variable
                            Scope nesting level: 1
                            Type form = subrange, Type id = range2
var4            044
                            Defined as: variable
                            Scope nesting level: 1
                            Type form = enumeration, Type id = enum1
var5            045
                            Defined as: variable
                            Scope nesting level: 1
                            Type form = subrange, Type id = <unnamed>
                            --- Base type ---
                            Type form = scalar, Type id = integer
                            Range = -7..10
var6            045
                            Defined as: variable
                            Scope nesting level: 1
                            Type form = subrange, Type id = <unnamed>
                            --- Base type ---
                            Type form = scalar, Type id = integer
                            Range = -7..10
var7            045
                            Defined as: variable
                            Scope nesting level: 1
                            Type form = subrange, Type id = <unnamed>
                            --- Base type ---
                            Type form = scalar, Type id = integer
                            Range = -7..10
var8            046
                            Defined as: variable
                            Scope nesting level: 1
                            Type form = enumeration, Type id = <unnamed>
                            --- Enumeration constants ---
                                        fee = 0
                                        fye = 1
                                        foe = 2
                                        fum = 3
var9            047
                            Defined as: variable
                            Scope nesting level: 1
                            Type form = subrange, Type id = range1
wednesday       017
                            Defined as: enumeration constant
                            Scope nesting level: 1
                            Type form = enumeration, Type id = week
                            Value = 2
week            017
                            Defined as: type
                            Scope nesting level: 1
                            Type form = enumeration, Type id = week
                            --- Enumeration constants ---
                                    monday = 0
                                    tuesday = 1
                                  wednesday = 2
                                   thursday = 3
                                     friday = 4
                                   saturday = 5
                                     sunday = 6
weekday         018
                            Defined as: type
                            Scope nesting level: 1
                            Type form = subrange, Type id = weekday
                            --- Base type ---
                            Type form = enumeration, Type id = week
                            Range = 0..4
weekend         019
                            Defined as: type
                            Scope nesting level: 1
                            Type form = subrange, Type id = weekend
                            --- Base type ---
                            Type form = enumeration, Type id = week
                            Range = 5..6
x               004
                            Defined as: constant
                            Scope nesting level: 1
                            Type form = scalar, Type id = char
                            Value = &apos;x&apos;

--- RECORD <unnamed> ---

Identifier      Line numbers    Type specification
----------      ------------    ------------------
a               038
                            Defined as: record field
                            Scope nesting level: 2
                            Type form = array, Type id = <unnamed>
r               037
                            Defined as: record field
                            Scope nesting level: 2
                            Type form = record, Type id = rec1
ten             036
                            Defined as: record field
                            Scope nesting level: 2
                            Type form = scalar, Type id = integer

--- RECORD rec1 ---

Identifier      Line numbers    Type specification
----------      ------------    ------------------
b1              031
                            Defined as: record field
                            Scope nesting level: 2
                            Type form = enumeration, Type id = boolean
b2              031
                            Defined as: record field
                            Scope nesting level: 2
                            Type form = enumeration, Type id = boolean
c               032
                            Defined as: record field
                            Scope nesting level: 2
                            Type form = scalar, Type id = char
i               029
                            Defined as: record field
                            Scope nesting level: 2
                            Type form = scalar, Type id = integer
r               030
                            Defined as: record field
                            Scope nesting level: 2
                            Type form = scalar, Type id = real

--- RECORD <unnamed> ---
```

```
Identifier      Line numbers    Type specification
----------      ------------    ------------------
a               059
                                Defined as: record field
                                Scope nesting level: 2
                                Type form = array, Type id = <unnamed>
b               051
                                Defined as: record field
                                Scope nesting level: 2
                                Type form = enumeration, Type id = boolean
r               052
                                Defined as: record field
                                Scope nesting level: 2
                                Type form = record, Type id = <unnamed>

                 0 statements executed.
                 0 runtime errors.
             0.00 seconds total execution time.
```

Listing 9-39 shows output from some syntax checking during declarations parsing.

Listing 9-39: Syntax errors while parsing declarations

```
001 CONST
002     ten = 10;
003     epsilon = 1.0E-6;
004     delta = epsilon/2;
                       ^
*** Unexpected token [at "/"]
005     pi = pi;
             ^
*** Not a constant identifier [at "pi"]
006
007 TYPE
008     typp = typp;
               ^
*** Not a type identifier [at "typp"]
009     range1 = 0..tenn;
                    ^
*** Undefined identifier [at "tenn"]
                    ^
*** Incompatible types [at "tenn"]
010     range3 = 0..10.0;
                    ^
*** Invalid subrange type [at "10.0"]
                    ^
*** Invalid subrange type [at "10.0"]
011     range5 = 'q'..'p';
                       ^
*** Min limit greater than max limit [at "'p'"]
012     range6 = foo..bar;
                  ^
*** Undefined identifier [at "foo"]
                     ^
*** Unexpected token [at ".."]
013
014     enum1 = (a, b, c, d, e)
015     range3 = e..c;
             ^
*** Missing ; [at "range3"]
             ^
*** Redefined identifier [at "range3"]
                    ^
*** Min limit greater than max limit [at "c"]
016
017     ar1 = ARRAY [integer] OF integer;
                      ^
*** Invalid index type [at "integer"]
018     ar4 = ARRAY [(foo, bar, baz)] OF (foo, bar);
                                          ^
*** Redefined identifier [at "foo"]
                                               ^
*** Redefined identifier [at "bar"]
019
020     rec1 = RECORD
021               i : integer;
022               r : real;
023               i : boolean;
                  ^
*** Redefined identifier [at "i"]
024            END;
025
026     ar5 = ARRAY [range2] RECORD
                     ^
*** Undefined identifier [at "range2"]
                            ^
*** Missing OF [at "RECORD"]
027                              rec : RECORD;
                                       ^
*** Missing END [at ";"]
028                              END;
029
030 VAR
031     var2 : range2;
               ^
*** Undefined identifier [at "range2"]
032     var4 : ten..5;
                   ^
*** Min limit greater than max limit [at "5"]
033     var5 : (fee, fye, foe, FYE, fum);
                                  ^
*** Redefined identifier [at "FYE"]
034
035 BEGIN
036 END.

                36 source lines.
                23 syntax errors.
             0.09 seconds total parsing time.
```

Make similar changes to the beginning of the utility class `ParseTreePrinter`. Listing 9-40 shows the modified `print()` method and the new method `printRoutine()`.

Listing 9-40: Methods `print()` and `printRoutine()` of class `ParseTreePrinter`

```
/**
 * Print the intermediate code as a parse tree.
 * @param symTabStack the symbol table stack.
 */
public void print(SymTabStack symTabStack)
{
    ps.println("\n===== INTERMEDIATE CODE =====");

    SymTabEntry programId = symTabStack.getProgramId();
    printRoutine(programId);
}

/**
 * Print the parse tree for a routine.
 * @param routineId the routine identifier&apos;s symbol table entry.
 */
private void printRoutine(SymTabEntry routineId)
{
    Definition definition = routineId.getDefinition();
    System.out.println("\n*** " + definition.toString() +
                       " " + routineId.getName() + " ***\n");

    // Print the intermediate code in the routine&apos;s symbol table entry.
    ICode iCode = (ICode) routineId.getAttribute(ROUTINE_ICODE);
    if (iCode.getRoot() != null) {
        printNode((ICodeNodeImpl) iCode.getRoot());
    }

    // Print any procedures and functions defined in the routine.
    ArrayList<SymTabEntry> routineIds =
        (ArrayList<SymTabEntry>) routineId.getAttribute(ROUTINE_ROUTINES);
    if (routineIds != null) {
        for (SymTabEntry rtnId : routineIds) {
            printRoutine(rtnId);
        }
    }
}
```

Method `print()` now takes a symbol table stack as its argument, since you're storing the intermediate code for each routine in the routine name's symbol table entry.

In Chapter 10, you will modify the Pascal statement parsers to handle array variables and record variables and to implement type checking.

Type Checking

In this chapter, you'll complete the work you started in Chapter 9 by incorporating type checking. Now that you can declare Pascal variables of various types, you need to ensure that when these variables appear in statements, their types must be compatible with their operators. As noted back in Chapter 1, type checking is part of semantic analysis.

Goals and Approach

The goal of this chapter is to incorporate type checking into the front end. The approach is to add type checking to the statement parsers so that they perform this task as they are parsing the Pascal statements.

A new version of the syntax checker that you've been developing since Chapter 5 will help to verify your work.

Type Checking

Begin with class `TypeChecker` in package `intermediate.typeimpl`. This class implements Pascal's type compatibility rules with static methods that the statement parsers use to perform type checking.

Listing 10-1 shows the methods that check for specific types.

Listing 10-1: Type checking methods in class `TypeChecker`

```
/**
 * Check if a type specification is integer.
 * @param type the type specification to check.
 * @return true if integer, else false.
 */
public static boolean isInteger(TypeSpec type)
{
    return (type != null) && (type.baseType() == Predefined.integerType);
}

/**
 * Check if both type specifications are integer.
 * @param type1 the first type specification to check.
 * @param type2 the second type specification to check.
 * @return true if both are integer, else false.
 */
public static boolean areBothInteger(TypeSpec type1, TypeSpec type2)
{
    return isInteger(type1) && isInteger(type2);
}

/**
 * Check if a type specification is real.
 * @param type the type specification to check.
 * @return true if real, else false.
 */
public static boolean isReal(TypeSpec type)
{
    return (type != null) && (type.baseType() == Predefined.realType);
}

/**
 * Check if a type specification is integer or real.
 * @param type the type specification to check.
 * @return true if integer or real, else false.
 */
public static boolean isIntegerOrReal(TypeSpec type)
{
    return isInteger(type) || isReal(type);
}

/**
 * Check if at least one of two type specifications is real.
 * @param type1 the first type specification to check.
 * @param type2 the second type specification to check.
 * @return true if at least one is real, else false.
 */
public static boolean isAtLeastOneReal(TypeSpec type1, TypeSpec type2)
{
    return (isReal(type1) && isReal(type2)) ||
           (isReal(type1) && isInteger(type2)) ||
           (isInteger(type1) && isReal(type2));
}

/**
 * Check if a type specification is boolean.
 * @param type the type specification to check.
 * @return true if boolean, else false.
 */
public static boolean isBoolean(TypeSpec type)
{
    return (type != null) && (type.baseType() == Predefined.booleanType);
}

/**
 * Check if both type specifications are boolean.
 * @param type1 the first type specification to check.
 * @param type2 the second type specification to check.
```

```
 * @return true if both are boolean, else false.
 */
public static boolean areBothBoolean(TypeSpec type1, TypeSpec type2)
{
    return isBoolean(type1) && isBoolean(type2);
}

/**
 * Check if a type specification is char.
 * @param type the type specification to check.
 * @return true if char, else false.
 */
public static boolean isChar(TypeSpec type)
{
    return (type != null) && (type.baseType() == Predefined.charType);
}
```

Listing 10-2 shows the type compatibility methods in class `TypeChecker`.

Listing 10-2: Type compatibility methods in class `TypeChecker`

```
/**
 * Check if two type specifications are assignment compatible.
 * @param targetType the target type specification.
 * @param valueType the value type specification.
 * @return true if the value can be assigned to the target, else false.
 */
public static boolean areAssignmentCompatible(TypeSpec targetType,
                                              TypeSpec valueType)
{
    if ((targetType == null) || (valueType == null)) {
        return false;
    }

    targetType = targetType.baseType();
    valueType  = valueType.baseType();

    boolean compatible = false;

    // Identical types.
    if (targetType == valueType) {
        compatible = true;
    }

    // real := integer
    else if (isReal(targetType) && isInteger(valueType)) {
        compatible = true;
    }

    // string := string
    else {
        compatible =
            targetType.isPascalString() && valueType.isPascalString();
    }

    return compatible;
}

/**
 * Check if two type specifications are comparison compatible.
 * @param type1 the first type specification to check.
 * @param type2 the second type specification to check.
 * @return true if the types can be compared to each other, else false.
 */
public static boolean areComparisonCompatible(TypeSpec type1,
                                              TypeSpec type2)
{
    if ((type1 == null) || (type2 == null)) {
        return false;
    }

    type1 = type1.baseType();
    type2 = type2.baseType();
    TypeForm form = type1.getForm();

    boolean compatible = false;

    // Two identical scalar or enumeration types.
    if ((type1 == type2) && ((form == SCALAR) || (form == ENUMERATION))) {
        compatible = true;
    }

    // One integer and one real.
    else if (isAtLeastOneReal(type1, type2)) {
        compatible = true;
    }

    // Two strings.
    else {
        compatible = type1.isPascalString() && type2.isPascalString();
    }

    return compatible;
}
```

Method `areAssignmentCompatible()` checks the type specifications of the target of an assignment statement (the left hand side) and the value (the right hand side). They are assignment compatible (and therefore the method returns true) if the two types are identical or if they are both strings or if the target type is real and the value type is integer.

Method `areComparisonCompatible()` checks the type specifications of two objects that will be compared to each other. They are comparison compatible if their base types are identical scalar or enumeration types or if they are both strings or if the base type of one is integer and the base type of the other is real.

Type Checking Expressions

Now that you have typed variables, expressions also become typed. As you parse expressions and subexpressions, record type information in the generated parse tree. Just as you added type specifications to the `SymTabEntry` objects in the previous chapter, add type specifications to the `ICodeNode` objects. Listing 10-3 shows the new setter and getter methods for interface `ICodeNode` in the `intermediate` package.

New methods `setTypeSpec()` and `getTypeSpec()` for interface `ICodeNode`

```
/**
 * Set the type specification of this node.
 * @param typeSpec the type sprcification to set.
 */
public void setTypeSpec(TypeSpec typeSpec);

/**
 * Return the type specification of this node.
 * @return the type specification.
 */
public TypeSpec getTypeSpec();
```

Design Note

Implementations of the new methods `setTypeSpec()` and `getTypeSpec()` allow the parsers to "decorate" the parse tree with type specification information. As you'll see, this enables doing type checking as part of semantic analysis.

These new methods are implemented straightforwardly in class `ICodeNodeImpl` in the `intermediate.icodeimpl` package, which now must also include the new field:

```
private TypeSpec typeSpec;    // data type specification
```

The statement parser subclass `ExpressionParser` now has new versions of methods `parseExpression()`, `parseSimpleExpression()`, `parseTerm()`, and `parseFactor()`, and a new method `parseIdentifier()`. Each method does type checking and sets the type specification of the root node of its generated parse subtree.

Listing 10-4 shows the new version of method `parseExpression()`.

Method `parseExpression()` of class `ExpressionParser`

```
/**
 * Parse an expression.
 * @param token the initial token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
private ICodeNode parseExpression(Token token)
    throws Exception
{
    // Parse a simple expression and make the root of its tree
    // the root node.
    ICodeNode rootNode = parseSimpleExpression(token);
    TypeSpec resultType = rootNode != null ? rootNode.getTypeSpec()
                                           : Predefined.undefinedType;

    token = currentToken();
    TokenType tokenType = token.getType();

    // Look for a relational operator.
    if (REL_OPS.contains(tokenType)) {

        // Create a new operator node and adopt the current tree
        // as its first child.
        ICodeNodeType nodeType = REL_OPS_MAP.get(tokenType);
        ICodeNode opNode = ICodeFactory.createICodeNode(nodeType);
        opNode.addChild(rootNode);

        token = nextToken();  // consume the operator

        // Parse the second simple expression.  The operator node adopts
        // the simple expression&apos;s tree as its second child.
        ICodeNode simExprNode = parseSimpleExpression(token);
        opNode.addChild(simExprNode);

        // The operator node becomes the new root node.
        rootNode = opNode;

        // Type check: The operands must be comparison compatible.
        TypeSpec simExprType = simExprNode != null
                                   ? simExprNode.getTypeSpec()
                                   : Predefined.undefinedType;
        if (TypeChecker.areComparisonCompatible(resultType, simExprType)) {
            resultType = Predefined.booleanType;
        }
        else {
            errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
            resultType = Predefined.undefinedType;
        }
    }

    if (rootNode != null) {
        rootNode.setTypeSpec(resultType);
    }

    return rootNode;
}
```

Method `parseExpression()` calls `parseSimpleExpression()` and sets variable `resultType` to the first simple expression's `TypeSpec` object. If there was a relational operator and a second simple expression, the method calls `TypeChecker.areComparisonCompatible()` to verify that the two operands are in fact comparison compatible. If all is well, the method sets the type specification of `rootNode` to the predefined boolean type. Otherwise, it sets the undefined type.

Listing 10-5 shows the new version of method `parseSimpleExpression()` with type checking.

Method `parseSimpleExpression()` of class `ExpressionParser`

```
/**
 * Parse a simple expression.
 * @param token the initial token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
private ICodeNode parseSimpleExpression(Token token)
```

```
        throws Exception
{
    Token signToken = null;
    TokenType signType = null;  // type of leading sign (if any)

    // Look for a leading + or - sign.
    TokenType tokenType = token.getType();
    if ((tokenType == PLUS) || (tokenType == MINUS)) {
        signType = tokenType;
        signToken = token;
        token = nextToken();  // consume the + or -
    }

    // Parse a term and make the root of its tree the root node.
    ICodeNode rootNode = parseTerm(token);
    TypeSpec resultType = rootNode != null ? rootNode.getTypeSpec()
                                           : Predefined.undefinedType;

    // Type check: Leading sign.
    if ((signType != null) && (!TypeChecker.isIntegerOrReal(resultType))) {
        errorHandler.flag(signToken, INCOMPATIBLE_TYPES, this);
    }

    // Was there a leading - sign?
    if (signType == MINUS) {

        // Create a NEGATE node and adopt the current tree
        // as its child. The NEGATE node becomes the new root node.
        ICodeNode negateNode = ICodeFactory.createICodeNode(NEGATE);
        negateNode.addChild(rootNode);
        negateNode.setTypeSpec(rootNode.getTypeSpec());
        rootNode = negateNode;
    }

    token = currentToken();
    tokenType = token.getType();

    // Loop over additive operators.
    while (ADD_OPS.contains(tokenType)) {
        TokenType operator = tokenType;

        // Create a new operator node and adopt the current tree
        // as its first child.
        ICodeNodeType nodeType = ADD_OPS_OPS_MAP.get(operator);
        ICodeNode opNode = ICodeFactory.createICodeNode(nodeType);
        opNode.addChild(rootNode);

        token = nextToken();  // consume the operator

        // Parse another term.  The operator node adopts
        // the term&apos;s tree as its second child.
        ICodeNode termNode = parseTerm(token);
        opNode.addChild(termNode);
        TypeSpec termType = termNode != null ? termNode.getTypeSpec()
                                             : Predefined.undefinedType;

        // The operator node becomes the new root node.
        rootNode = opNode;

        // Determine the result type.
        switch ((PascalTokenType) operator) {

            case PLUS:
            case MINUS: {
                // Both operands integer ==> integer result.
                if (TypeChecker.areBothInteger(resultType, termType)) {
                    resultType = Predefined.integerType;
                }

                // Both real operands or one real and one integer operand
                // ==> real result.
                else if (TypeChecker.isAtLeastOneReal(resultType,
                                                      termType)) {
                    resultType = Predefined.realType;
                }

                else {
                    errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
                }

                break;
            }

            case OR: {
                // Both operands boolean ==> boolean result.
                if (TypeChecker.areBothBoolean(resultType, termType)) {
                    resultType = Predefined.booleanType;
                }
                else {
                    errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
                }

                break;
            }
        }

        rootNode.setTypeSpec(resultType);

        token = currentToken();
        tokenType = token.getType();
    }

    return rootNode;
}
```

If there was a leading + or - sign, `parseSimpleExpression()` calls `TypeChecker.isIntegerOrReal()` to check the type of the expression. For the binary + or - operator parsed in the `switch` statement, calls to `TypeChecker.areBothInteger()` and `TypeChecker.isAtLeastOneReal()` determine the result type. A call to `TypeChecker.areBothBoolean()` checks both operands of the `OR` operator. The method sets the result type specification into the expression tree's root node.

shows the corresponding `switch` statement in the new version of method `parseTerm()`.

Listing 10-6: The **`switch`** statement in method **`parseTerm()`** of class **`ExpressionParser`**

```
        // Determine the result type.
        switch ((PascalTokenType) operator) {

            case STAR: {
```

```
                    // Both operands integer ==> integer result.
                    if (TypeChecker.areBothInteger(resultType, factorType)) {
                        resultType = Predefined.integerType;
                    }

                    // Both real operands or one real and one integer operand
                    // ==> real result.
                    else if (TypeChecker.isAtLeastOneReal(resultType,
                                                          factorType)) {
                        resultType = Predefined.realType;
                    }

                    else {
                        errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
                    }

                    break;
                }

                case SLASH: {
                    // All integer and real operand combinations
                    // ==> real result.
                    if (TypeChecker.areBothInteger(resultType, factorType) ||
                        TypeChecker.isAtLeastOneReal(resultType, factorType))
                    {
                        resultType = Predefined.realType;
                    }
                    else {
                        errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
                    }

                    break;
                }

                case DIV:
                case MOD: {
                    // Both operands integer ==> integer result.
                    if (TypeChecker.areBothInteger(resultType, factorType)) {
                        resultType = Predefined.integerType;
                    }
                    else {
                        errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
                    }

                    break;
                }

                case AND: {
                    // Both operands boolean ==> boolean result.
                    if (TypeChecker.areBothBoolean(resultType, factorType)) {
                        resultType = Predefined.booleanType;
                    }
                    else {
                        errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
                    }

                    break;
                }
            }
        }
```

For the `*` operator, a call to `TypeChecker.areBothInteger()` or to `TypeChecker.isAtLeastOneReal()` determines whether the result type ought to be integer or real, respectively. Similar calls are made for the `/` operator but the result is always real. For the DIV and MOD operators, a call to `TypeChecker.areBothInteger()` verifies that both operands are integer and the result is always integer. Finally, if the operator is AND, `TypeChecker.areBothBoolean()` is called and the result is always boolean.

The new version of method `parseFactor()` also has changes in its `switch` statement. See Listing 10-7 for some examples.

Listing 10-7: Some of the cases in the `switch` statement of method `parseFactor()` of class `ExpressionParser`

```
case IDENTIFIER: {
    return parseIdentifier(token);
}

case INTEGER: {
    // Create an INTEGER_CONSTANT node as the root node.
    rootNode = ICodeFactory.createICodeNode(INTEGER_CONSTANT);
    rootNode.setAttribute(VALUE, token.getValue());

    token = nextToken();  // consume the number

    rootNode.setTypeSpec(Predefined.integerType);
    break;
}

case STRING: {
    String value = (String) token.getValue();

    // Create a STRING_CONSTANT node as the root node.
    rootNode = ICodeFactory.createICodeNode(STRING_CONSTANT);
    rootNode.setAttribute(VALUE, value);

    TypeSpec resultType = value.length() == 1
                              ? Predefined.charType
                              : TypeFactory.createStringType(value);

    token = nextToken();  // consume the string

    rootNode.setTypeSpec(resultType);
    break;
}

case NOT: {
    token = nextToken();  // consume the NOT

    // Create a NOT node as the root node.
    rootNode = ICodeFactory.createICodeNode(ICodeNodeTypeImpl.NOT);

    // Parse the factor.  The NOT node adopts the
    // factor node as its child.
    ICodeNode factorNode = parseFactor(token);
    rootNode.addChild(factorNode);

    // Type check: The factor must be boolean.
    TypeSpec factorType = factorNode != null
                              ? factorNode.getTypeSpec()
                              : Predefined.undefinedType;
    if (!TypeChecker.isBoolean(factorType)) {
        errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
```

```
                }
                rootNode.setTypeSpec(Predefined.booleanType);
                break;
            }
```

The `IDENTIFIER` case now calls a new method `parseIdentifier()`. The `INTEGER` case sets the type specification of the `INTEGER_CONSTANT` node to the predefined integer type. (The `REAL` case, not shown, is similar.) The `STRING` case must decide between the predefined character type or a string type. The `NOT` case calls `TypeChecker.isBoolean()` to verify that the type of the expression is boolean and sets the type specification of the `NOT` node to the predefined boolean type. The `LEFT_PAREN` case (not shown) simply uses the type specification of the nested expression.

[Listing 10-8](#) shows the new `parseIdentifier()` method.

[Listing 10-8:](#) Method `parseIdentifier()` of class `ExpressionParser`

```
/**
 * Parse an identifier.
 * @param token the current token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
private ICodeNode parseIdentifier(Token token)
    throws Exception
{
    ICodeNode rootNode = null;

    // Look up the identifier in the symbol table stack.
    String name = token.getText().toLowerCase();
    SymTabEntry id = symTabStack.lookup(name);

    // Undefined.
    if (id == null) {
        errorHandler.flag(token, IDENTIFIER_UNDEFINED, this);
        id = symTabStack.enterLocal(name);
        id.setDefinition(UNDEFINED);
        id.setTypeSpec(Predefined.undefinedType);
    }

    Definition defnCode = id.getDefinition();

    switch ((DefinitionImpl) defnCode) {

        case CONSTANT: {
            Object value = id.getAttribute(CONSTANT_VALUE);
            TypeSpec type = id.getTypeSpec();

            if (value instanceof Integer) {
                rootNode = ICodeFactory.createICodeNode(INTEGER_CONSTANT);
                rootNode.setAttribute(VALUE, value);
            }
            else if (value instanceof Float) {
                rootNode = ICodeFactory.createICodeNode(REAL_CONSTANT);
                rootNode.setAttribute(VALUE, value);
            }
            else if (value instanceof String) {
                rootNode = ICodeFactory.createICodeNode(STRING_CONSTANT);
                rootNode.setAttribute(VALUE, value);
            }

            id.appendLineNumber(token.getLineNumber());
            token = nextToken();  // consume the constant identifier

            if (rootNode != null) {
                rootNode.setTypeSpec(type);
            }

            break;
        }

        case ENUMERATION_CONSTANT: {
            Object value = id.getAttribute(CONSTANT_VALUE);
            TypeSpec type = id.getTypeSpec();

            rootNode = ICodeFactory.createICodeNode(INTEGER_CONSTANT);
            rootNode.setAttribute(VALUE, value);

            id.appendLineNumber(token.getLineNumber());
            token = nextToken();  // consume the enum constant identifier

            rootNode.setTypeSpec(type);
            break;
        }

        default: {
            VariableParser variableParser = new VariableParser(this);
            rootNode = variableParser.parse(token, id);
            break;
        }
    }

    return rootNode;
}
```

Method `parseIdentifier()` first verifies that the identifier is defined locally or in an enclosing scope (lower numbered nesting level). If not, the method enters the identifier into the local symbol table as an undefined identifier with an undefined type.

If the identifier is a constant identifier, then `parseIdentifier()` creates either an `INTEGER_CONSTANT`, a `REAL_CONSTANT`, or a `STRING_CONSTANT` node depending on the type of the constant value, and it sets the value of the node's `VALUE` attribute. The method treats an identifier that is an enumeration constant similarly, except that it always creates an `INTEGER_CONSTANT` node.
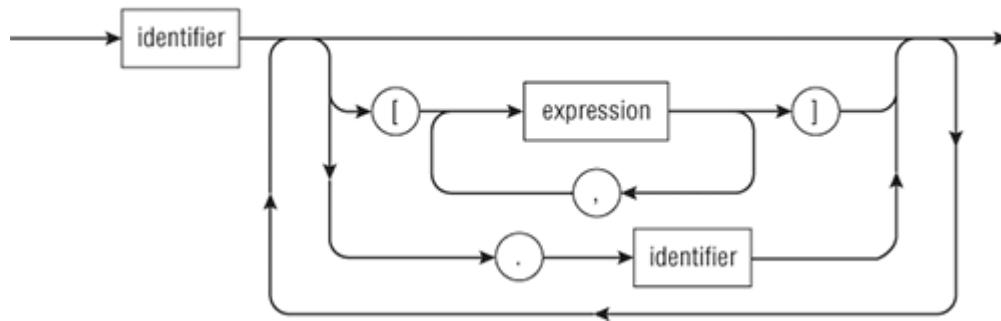
If the identifier is neither a constant identifier nor an enumeration constant, then `parseIdentifier()` calls `variableParser.parse()` to parse a variable.

Variables

[Figure 10-1](#) shows the complete syntax diagram for a Pascal variable, which can be the target of an assignment statement or appear in an expression. [Figure 10-1](#) expands upon the definition of a variable in Figure 5-1. A variable can include multiple subscripts and fields.

Figure 10-1: The complete syntax diagram for a variable. The outermost loop back indicates that a variable can include multiple subscripts and fields.

variable

[Listing 10-9](#) shows the two `parse()` methods of the new statement parser subclass `VariableParser` in package `frontend.pascal.parsers`.

Listing 10-9: The `parse()` methods of class `VariableParser`

```
// Synchronization set to start a subscript or a field.
private static final EnumSet<PascalTokenType> SUBSCRIPT_FIELD_START_SET =
    EnumSet.of(LEFT_BRACKET, DOT);

/**
 * Parse a variable.
 * @param token the initial token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token)
    throws Exception
{
    // Look up the identifier in the symbol table stack.
    String name = token.getText().toLowerCase();
    SymTabEntry variableId = symTabStack.lookup(name);

    // If not found, flag the error and enter the identifier
    // as an undefined identifier with an undefined type.
    if (variableId == null) {
        errorHandler.flag(token, IDENTIFIER_UNDEFINED, this);
        variableId = symTabStack.enterLocal(name);
        variableId.setDefinition(UNDEFINED);
        variableId.setTypeSpec(Predefined.undefinedType);
    }

    return parse(token, variableId);
}

/**
 * Parse a variable.
 * @param token the initial token.
 * @param variableId the symbol table entry of the variable identifier.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token, SymTabEntry variableId)
    throws Exception
{
    // Check how the variable is defined.
    Definition defnCode = variableId.getDefinition();
    if ((defnCode != VARIABLE) && (defnCode != VALUE_PARM) &&
        (defnCode != VAR_PARM))
    {
        errorHandler.flag(token, INVALID_IDENTIFIER_USAGE, this);
    }

    variableId.appendLineNumber(token.getLineNumber());

    ICodeNode variableNode =
        ICodeFactory.createICodeNode(ICodeNodeTypeImpl.VARIABLE);
    variableNode.setAttribute(ID, variableId);

    token = nextToken();  // consume the identifier

    // Parse array subscripts or record fields.
    TypeSpec variableType = variableId.getTypeSpec();
    while (SUBSCRIPT_FIELD_START_SET.contains(token.getType())) {
        ICodeNode subFldNode = token.getType() == LEFT_BRACKET
                                   ? parseSubscripts(variableType)
                                   : parseField(variableType);
        token = currentToken();

        // Update the variable&apos;s type.
        // The variable node adopts the SUBSCRIPTS or FIELD node.
        variableType = subFldNode.getTypeSpec();
        variableNode.addChild(subFldNode);
    }

    variableNode.setTypeSpec(variableType);
    return variableNode;
}
```

The first `parse()` method is an override of the `parse()` method of the `StatementParser` superclass. It looks up the identifier (the current token) in the symbol table stack and if the identifier is not found, it enters the identifier into the local symbol table as an undefined identifier with an undefined type. The method then calls the second `parse()` method, passing both the token and the symbol table entry.

The second `parse()` method checks how the identifier is defined before creating a `VARIABLE` parse tree node. Each iteration of the `while` loop calls method `parseSubscripts()` or method `parseField()` to parse any subscripts or record fields, respectively, depending on whether the current token is a left bracket or a period. Each iteration updates variable `variableType` to the current `TypeSpec` object. The final value of `variableType` will be the `TypeSpec` object of the last array element or record field. The `VARIABLE` node adopts the `SUBSCRIPTS` or `FIELD` nodes generated by these methods.

shows method `parseSubscripts()`.

Listing 10-10: Method **`parseSubscripts()`** of class **`VariableParser`**

```
    // Synchronization set for the ] token.
    private static final EnumSet<PascalTokenType> RIGHT_BRACKET_SET =
        EnumSet.of(RIGHT_BRACKET, EQUALS, SEMICOLON);

    /**
     * Parse a set of comma-separated subscript expressions.
     * @param variableType the type of the array variable.
     * @return the root node of the generated parse tree.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseSubscripts(TypeSpec variableType)
        throws Exception
    {
        Token token;
        ExpressionParser expressionParser = new ExpressionParser(this);

        // Create a SUBSCRIPTS node.
        ICodeNode subscriptsNode = ICodeFactory.createICodeNode(SUBSCRIPTS);

        do {
            token = nextToken();  // consume the [ or , token

            // The current variable is an array.
            if (variableType.getForm() == ARRAY) {

                // Parse the subscript expression.
                ICodeNode exprNode = expressionParser.parse(token);
                TypeSpec exprType = exprNode != null ? exprNode.getTypeSpec()
                                                     : Predefined.undefinedType;

                // The subscript expression type must be assignment
                // compatible with the array index type.
                TypeSpec indexType =
                    (TypeSpec) variableType.getAttribute(ARRAY_INDEX_TYPE);
                if (!TypeChecker.areAssignmentCompatible(indexType, exprType)) {
                    errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
                }

                // The SUBSCRIPTS node adopts the subscript expression tree.
                subscriptsNode.addChild(exprNode);

                // Update the variable&apos;s type.
                variableType =
                    (TypeSpec) variableType.getAttribute(ARRAY_ELEMENT_TYPE);
            }

            // Not an array type, so too many subscripts.
            else {
                errorHandler.flag(token, TOO_MANY_SUBSCRIPTS, this);
                expressionParser.parse(token);
            }

            token = currentToken();
        } while (token.getType() == COMMA);

        // Synchronize at the ] token.
        token = synchronize(RIGHT_BRACKET_SET);
        if (token.getType() == RIGHT_BRACKET) {
            token = nextToken();  // consume the ] token
        }
        else {
            errorHandler.flag(token, MISSING_RIGHT_BRACKET, this);
        }

        subscriptsNode.setTypeSpec(variableType);
        return subscriptsNode;
    }
```

Method `parseSubscripts()` parses a set of one or more comma-separated subscripts enclosed in square brackets. The initial value of argument `variableType` is the current `TypeSpec` object of the variable as it has been parsed thus far. The method creates a `SUBSCRIPTS` node and verifies that the current type is an array. It loops to call `expressionParser.parse()` to parse each subscript expression and the `SUBSCRIPTS` node adopts each expression subtree. The method calls `TypeChecker.areAssignmentCompatible()` to check whether each subscript expression is assignment compatible with the corresponding index type, and it also checks that there aren't too many subscripts for the array type. It updates `variableType` as it parses each subscript, and it sets the type specification of the `SUBSCRIPTS` node to the final value of `variableType`.

shows method `parseField()`.

Listing 10-11: Method **`parseField()`** of class **`VariableParser`**

```
    /**
     * Parse a record field.
     * @param variableType the type of the record variable.
     * @return the root node of the generated parse tree.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseField(TypeSpec variableType)
        throws Exception
    {
        // Create a FIELD node.
        ICodeNode fieldNode = ICodeFactory.createICodeNode(FIELD);

        Token token = nextToken();  // consume the . token
        TokenType tokenType = token.getType();
        TypeForm variableForm = variableType.getForm();

        if ((tokenType == IDENTIFIER) && (variableForm == RECORD)) {
            SymTab symTab = (SymTab) variableType.getAttribute(RECORD_SYMTAB);
            String fieldName = token.getText().toLowerCase();
            SymTabEntry fieldId = symTab.lookup(fieldName);

            if (fieldId != null) {
                variableType = fieldId.getTypeSpec();
                fieldId.appendLineNumber(token.getLineNumber());

                // Set the field identifier&apos;s name.
                fieldNode.setAttribute(ID, fieldId);
            }
            else {
```

```
           errorHandler.flag(token, INVALID_FIELD, this);
        }
    }
    else {
        errorHandler.flag(token, INVALID_FIELD, this);
    }

    token = nextToken();  // consume the field identifier

    fieldNode.setTypeSpec(variableType);
    return fieldNode;
}
```

Method `parseField()` parses a single record field. It also has a `variableType` argument whose initial value is the current `TypeSpec` object of the variable as it has been parsed thus far. The method creates a FIELD node, verifies that the current type is a record, and looks up the field identifier in the record type's symbol table. If it finds the identifier, the method sets the field identifier's name into the FIELD node. It sets the type specification of the FIELD node to the type specification of the field identifier.

Figure 10-2 shows the parse tree for an assignment statement with a target variable that has subscripts and fields.
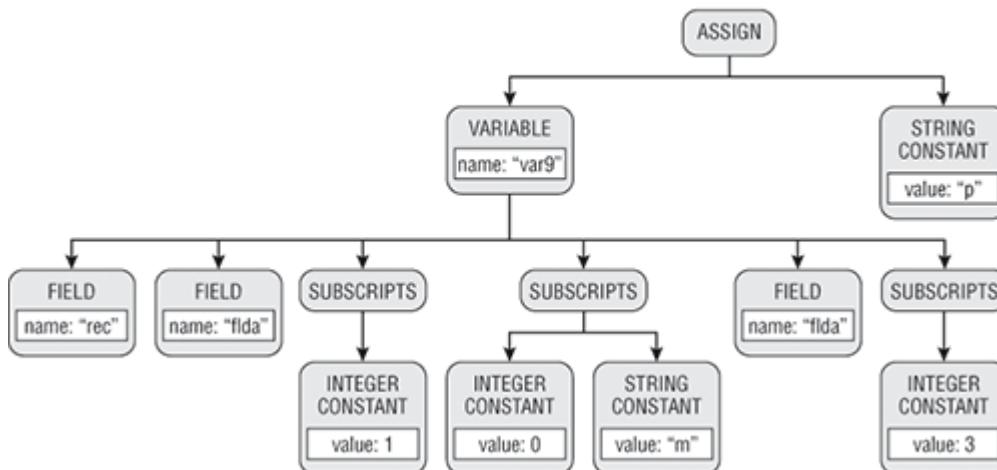
Type Checking Control Statements

Since Pascal control statements contain expressions, their parsers also need to do type checking.

Listing 10-12 shows a new version of the `parse()` method of the statement parser subclass `AssignmentStatementParser`.

Figure 10-2: The parse tree for an assignment statement with a target variable that has subscripts and fields. Identifier b is an enumeration constant whose value is 1 and identifier d is an enumeration constant whose value is 3.

```
var9.rec.flda[b][0, 'm'].flda[d] := 'p'
```


Listing 10-12: Method `parse()` of class `AssignmentStatementParser`

```
/**
 * Parse an assignment statement.
 * @param token the initial token.
 * @return intermediate info containing the root node of the
 * generated parse tree and the type specification.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token)
    throws Exception
{
    // Create the ASSIGN node.
    ICodeNode assignNode = ICodeFactory.createICodeNode(ASSIGN);

    // Parse the target variable.
    VariableParser variableParser = new VariableParser(this);
    ICodeNode targetNode = variableParser.parse(token);
    TypeSpec targetType = targetNode != null ? targetNode.getTypeSpec()
                                             : Predefined.undefinedType;

    // The ASSIGN node adopts the variable node as its first child.
    assignNode.addChild(targetNode);

    // Synchronize on the := token.
    token = synchronize(COLON_EQUALS_SET);
    if (token.getType() == COLON_EQUALS) {
        token = nextToken();  // consume the :=
    }
    else {
        errorHandler.flag(token, MISSING_COLON_EQUALS, this);
    }

    // Parse the expression.  The ASSIGN node adopts the expression's
    // node as its second child.
    ExpressionParser expressionParser = new ExpressionParser(this);
    ICodeNode exprNode = expressionParser.parse(token);
    assignNode.addChild(exprNode);

    // Type check: Assignment compatible?
    TypeSpec exprType = exprNode != null ? exprNode.getTypeSpec()
                                         : Predefined.undefinedType;
    if (!TypeChecker.areAssignmentCompatible(targetType, exprType)) {
        errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
    }

    assignNode.setTypeSpec(targetType);
    return assignNode;
}
```

The `parse()` method calls `variableParser.parse()` to parse the target variable and `TypeChecker.areAssignmentCompatible()` to check whether the type of the target variable is assignment compatible with the type of the expression value. It sets the type specification of the `ASSIGN` node to the type specification of the target variable.

The new version of the `parse()` method of class `RepeatStatementParser` also does type checking. It calls `TypeChecker.isBoolean()` to verify that the expression is type boolean. See [Listing 10-13](#).

Listing 10-13: Type checking in method `parse()` of class `RepeatStatementParser`

```
// Parse the expression.
// The TEST node adopts the expression subtree as its only child.
ExpressionParser expressionParser = new ExpressionParser(this);
ICodeNode exprNode = expressionParser.parse(token);
testNode.addChild(exprNode);

// Type check: The test expression must be boolean.
TypeSpec exprType = exprNode != null ? exprNode.getTypeSpec()
                                     : Predefined.undefinedType;
if (!TypeChecker.isBoolean(exprType)) {
    errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
}
```

The new version of the `parse()` method of class `WhileStatementParser` is similar. It also calls `TypeChecker.isBoolean()` to verify that its expression type is boolean. See [Listing 10-14](#).

Listing 10-14: Type checking in method `parse()` of class `WhileStatementParser`

```
// Parse the expression.
// The NOT node adopts the expression subtree as its only child.
ExpressionParser expressionParser = new ExpressionParser(this);
ICodeNode exprNode = expressionParser.parse(token);
notNode.addChild(exprNode);

// Type check: The test expression must be boolean.
TypeSpec exprType = exprNode != null ? exprNode.getTypeSpec()
                                     : Predefined.undefinedType;
if (!TypeChecker.isBoolean(exprType)) {
    errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
}
```

Listings [10-15](#), [10-16](#), and [10-17](#) show type checking in new versions of the `parse()` methods of statement parser subclasses `ForStatementParser`, `IfStatementParser`, and `CaseStatementParser`, respectively.

Listing 10-15: Type checking in method `parse()` of class `ForStatementParser`

```
// Parse the embedded initial assignment.
AssignmentStatementParser assignmentParser =
    new AssignmentStatementParser(this);
ICodeNode initAssignNode = assignmentParser.parse(token);
TypeSpec controlType = initAssignNode != null
                       ? initAssignNode.getTypeSpec()
                       : Predefined.undefinedType;

// Type check: The control variable&apos;s type must be integer
//             or enumeration.
if (!TypeChecker.isInteger(controlType) &&
    (controlType.getForm() != ENUMERATION))
{
    errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
}

// The COMPOUND node adopts the initial ASSIGN and the LOOP nodes
// as its first and second children.
compoundNode.addChild(initAssignNode);
compoundNode.addChild(loopNode);

// Synchronize at the TO or DOWNTO.
token = synchronize(TO_DOWNTO_SET);
TokenType direction = token.getType();

// Look for the TO or DOWNTO.
if ((direction == TO) || (direction == DOWNTO)) {
    token = nextToken();  // consume the TO or DOWNTO
}
else {
    direction = TO;
    errorHandler.flag(token, MISSING_TO_DOWNTO, this);
}

// Create a relational operator node: GT for TO, or LT for DOWNTO.
ICodeNode relOpNode = ICodeFactory.createICodeNode(direction == TO
                                                   ? GT : LT);
relOpNode.setTypeSpec(Predefined.booleanType);

// Copy the control VARIABLE node. The relational operator
// node adopts the copied VARIABLE node as its first child.
ICodeNode controlVarNode = initAssignNode.getChildren().get(0);
relOpNode.addChild(controlVarNode.copy());

// Parse the termination expression. The relational operator node
// adopts the expression as its second child.
ExpressionParser expressionParser = new ExpressionParser(this);
ICodeNode exprNode = expressionParser.parse(token);
relOpNode.addChild(exprNode);

// Type check: The termination expression type must be assignment
//             compatible with the control variable&apos;s type.
TypeSpec exprType = exprNode != null ? exprNode.getTypeSpec()
                                     : Predefined.undefinedType;
if (!TypeChecker.areAssignmentCompatible(controlType, exprType)) {
    errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
}
```

The `FOR` control variable's type must be integer, character, or enumeration. The `parse()` method calls `assignmentParser.parse()`, which performs type checking between the control variable and the initial expression. It calls `TypeChecker.areAssignmentCompatible()` to verify that the termination expression's type is assignment compatible with the control variable's type.

```
        // Parse the expression.
        // The IF node adopts the expression subtree as its first child.
        ExpressionParser expressionParser = new ExpressionParser(this);
        ICodeNode exprNode = expressionParser.parse(token);
        ifNode.addChild(exprNode);

        // Type check: The expression type must be boolean.
        TypeSpec exprType = exprNode != null ? exprNode.getTypeSpec()
                                             : Predefined.undefinedType;
        if (!TypeChecker.isBoolean(exprType)) {
            errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
        }
```

The new version of the `parse()` method of the `CaseStatementParser` must verify that the case expression type is either integer, character, or enumeration.

```
        // Parse the CASE expression.
        // The SELECT node adopts the expression subtree as its first child.
        ExpressionParser expressionParser = new ExpressionParser(this);
        ICodeNode exprNode = expressionParser.parse(token);
        selectNode.addChildNode(exprNode);

        // Type check: The CASE expression&apos;s type must be integer, character,
        //             or enumeration.
        TypeSpec exprType = exprNode.getTypeSpec();
        if (!TypeChecker.isInteger(exprType) &&
            !TypeChecker.isChar(exprType) &&
            (exprType.getForm() != ENUMERATION))
        {
            errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
        }
```

Method `parseConstant()` now sets the type specification of the constant node. It calls `TypeChecker.areComparisonCompatible()` to check whether each constant value is comparison compatible with the case expression's type. See .

```
        // Parse the constant.
        switch ((PascalTokenType) token.getType()) {

            case IDENTIFIER: {
                constantNode = parseIdentifierConstant(token, sign);
                if (constantNode != null) {
                    constantType = constantNode.getTypeSpec();
                }

                break;
            }

            case INTEGER: {
                constantNode = parseIntegerConstant(token.getText(), sign);
                constantType = Predefined.integerType;
                break;
            }

            case STRING: {
                constantNode =
                    parseCharacterConstant(token, (String) token.getValue(),
                                           sign);
                constantType = Predefined.charType;
                break;
            }

            default: {
                errorHandler.flag(token, INVALID_CONSTANT, this);
                break;
            }
        }

        // Check for reused constants.
        if (constantNode != null) {
            Object value = constantNode.getAttribute(VALUE);

            if (constantSet.contains(value)) {
                errorHandler.flag(token, CASE_CONSTANT_REUSED, this);
            }
            else {
                constantSet.add(value);
            }
        }

        // Type check: The constant type must be comparison compatible
        //             with the CASE expression type.
        if (!TypeChecker.areComparisonCompatible(expressionType,
                                                 constantType)) {
            errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
        }

        token = nextToken();  // consume the constant

        constantNode.setTypeSpec(constantType) ;
        return constantNode;
```

Method `parseIdentifierConstant()` can now parse a case branch constant that is an identifier. See .

```
    /**
     * Parse an identifier CASE constant.
     * @param value the current token value string.
     * @param sign the sign, if any.
     * @return the constant node.
     */
    private ICodeNode parseIdentifierConstant(Token token, TokenType sign)
        throws Exception
    {
        ICodeNode constantNode = null;
        TypeSpec constantType = null;
```

```
        // Look up the identifier in the symbol table stack.
        String name = token.getText().toLowerCase();
        SymTabEntry id = symTabStack.lookup(name);

        // Undefined.
        if (id == null) {
            id = symTabStack.enterLocal(name);
            id.setDefinition(UNDEFINED);
            id.setTypeSpec(Predefined.undefinedType);
            errorHandler.flag(token, IDENTIFIER_UNDEFINED, this);
            return null;
        }

        Definition defnCode = id.getDefinition();

        // Constant identifier.
        if ((defnCode == CONSTANT) || (defnCode == ENUMERATION_CONSTANT)) {
            Object constantValue = id.getAttribute(CONSTANT_VALUE);
            constantType = id.getTypeSpec();

            // Type check: Leading sign permitted only for integer constants.
            if ((sign != null) && !TypeChecker.isInteger(constantType)) {
                errorHandler.flag(token, INVALID_CONSTANT, this);
            }

            constantNode = ICodeFactory.createICodeNode(INTEGER_CONSTANT);
            constantNode.setAttribute(VALUE, constantValue);
        }

        id.appendLineNumber(token.getLineNumber());

        if (constantNode != null) {
            constantNode.setTypeSpec(constantType);
        }

        return constantNode;
    }
```

Method `parseIdentifierConstant()` verifies that the identifier is a constant or an enumeration constant, and that whenever there is a leading + or – sign before a constant identifier, the type of the identifier must be an integer.

Another change for the CASE statement, this time in the interpreter back end. Listing 10-20 shows the new version of method `createJumpTable()` of class `SelectExecutor`. For a CASE statement where the expression value is a character, the method converts each CASE branch constant value from a string (it will contain only one character) to a character before entering it into the jump table as a key.

Listing 10-20: New version of method `createJumpTable()` of class `SelectExecutor`

```
/**
 * Create a jump table for a SELECT node.
 * @param node the SELECT node.
 * @return the jump table.
 */
private HashMap<Object, ICodeNode> createJumpTable(ICodeNode node)
{
    HashMap<Object, ICodeNode> jumpTable = new HashMap<Object, ICodeNode>();

    // Loop over children that are SELECT_BRANCH nodes.
    ArrayList<ICodeNode> selectChildren = node.getChildren();
    for (int i = 1; i < selectChildren.size(); ++i) {
        ICodeNode branchNode = selectChildren.get(i);
        ICodeNode constantsNode = branchNode.getChildren().get(0);
        ICodeNode statementNode = branchNode.getChildren().get(1);

        // Loop over the constants children of the branch&apos;s CONSTANTS_NODE.
        ArrayList<ICodeNode> constantsList = constantsNode.getChildren();
        for (ICodeNode constantNode : constantsList) {

            // Create a jump table entry.
            // Convert a single-character string constant to a character.
            Object value = constantNode.getAttribute(VALUE);
            if (constantNode.getType() == STRING_CONSTANT) {
                value = ((String) value).charAt(0);
            }
            jumpTable.put(value, statementNode);
        }
    }

    return jumpTable;
}
```

Program 10: Pascal Syntax Checker III

Syntax checking now includes type checking, and the parser can generate parse trees for variables with subscripts and fields.
Listing 10-21 is the output from a command line similar to:

```
java -classpath classes Pascal compile -i block.txt
```

The command is to compile rather than execute because you haven't written the executors for variables yet. You'll do that in Chapter 12.

Listing 10-21: Output from the Pascal syntax checker with type checking

```
001 CONST
002     seven =  7;
003     ten   = 10;
004
005 TYPE
006     range1 = 0..ten;
007     range2 = &apos;a&apos;..&apos;q&apos;;
008     range3 = range1;
009
010     enum1 = (a, b, c, d, e);
011     enum2 = enum1;
012
013     range4 = b..d;
014
```

```
015     arr1 = ARRAY [range1] OF real;
016     arr2 = ARRAY [(alpha, beta, gamma)] OF range2;
017     arr3 = ARRAY [enum2] OF arr1;
018     arr4 = ARRAY [range3] OF (foo, bar, baz);
019     arr5 = ARRAY [range1] OF ARRAY[range2] OF ARRAY[c..e] OF enum2;
020     arr6 = ARRAY [range1, range2, c..e] OF enum2;
021
022     rec7 = RECORD
023              i : integer;
024              r : real;
025              b1, b2 : boolean;
026              c : char
027            END;
028
029     arr8 = ARRAY [range2] OF RECORD
030                                fldi  : integer;
031                                fldr : rec7;
032                                flda : ARRAY[range4] OF range2;
033                              END;
034
035 VAR
036     var1 : arr1;  var5 : arr5;
037     var2 : arr2;  var6 : arr6;
038     var3 : arr3;  var7 : rec7;
039     var4 : arr4;  var8 : arr8;
040
041     var9 : RECORD
042              b   : boolean;
043              rec : RECORD
044                      fld1 : arr1;
045                      fldb : boolean;
046                      fldr : real;
047                      fld6 : arr6;
048                      flda : ARRAY [enum1, range1] OF arr8;
049                    END;
050              a : ARRAY [1..5] OF boolean;
051            END;
052
053 BEGIN
054     var1[5] := 3.14;
055     var1[var7.i] := var9.rec.flda[e, ten][&apos;q&apos;].fldi;
056
057     IF var9.a[seven-3] THEN var2[beta] := &apos;x&apos;;;
058
059     CASE var4[var8[&apos;m&apos;].fldi - 4] OF
060        foo:      var3[e][3] := 12;
061        bar, baz: var3[b] := var1;
062     END;
063
064     REPEAT
065         var5[3] := var5[4];
066         var5[3, &apos;a&apos;] := var5[4][&apos;f&apos;];
067         var5[3, &apos;a&apos;, c] := var6[4, &apos;f&apos;][d];
068     UNTIL var6[3, &apos;a&apos;, c] > var5[4][&apos;f&apos;, d];
069
070     WHILE var7.i <> var9.rec.fldr DO BEGIN
071         var7.r := var8[&apos;g&apos;].fldr.r;
072     END;
073
074     var6[3] := var6[4];
075     var6[3, &apos;a&apos;] := var6[4][&apos;f&apos;];
076
077     var9.rec.fld6[4][&apos;f&apos;][d] := e;
078     var9.rec.fld6[4, &apos;f&apos;][d] := e;
079     var9.rec.flda[b, 0, &apos;m&apos;].flda[d] := &apos;p&apos;;;
080     var9.rec.flda[b][0, &apos;m&apos;].flda[d] := &apos;p&apos;;;
081 END.

                81 source lines.
                 0 syntax errors.
               0.13 seconds total parsing time.

===== INTERMEDIATE CODE =====

*** PROGRAM dummyprogramname ***

<COMPOUND line="53">
    <ASSIGN line="54" type_id="real">
        <VARIABLE id="var1" level="1" type_id="real">
            <SUBSCRIPTS type_id="real">
                <INTEGER_CONSTANT value="5" type_id="integer" />
            </SUBSCRIPTS>
        </VARIABLE>
        <REAL_CONSTANT value="3.14" type_id="real" />
    </ASSIGN>
    <ASSIGN line="55" type_id="real">
        <VARIABLE id="var1" level="1" type_id="real">
            <SUBSCRIPTS type_id="real">
                <VARIABLE id="var7" level="1" type_id="integer">
                    <FIELD id="i" level="2" type_id="integer" />
                </VARIABLE>
            </SUBSCRIPTS>
        </VARIABLE>
        <VARIABLE id="var9" level="1" type_id="integer">
            <FIELD id="rec" level="2" type_id="$anon_16e7c77" />
            <FIELD id="flda" level="3" type_id="$anon_b6ae9766" />
            <SUBSCRIPTS type_id="arr8">
                <INTEGER_CONSTANT value="4" type_id="enum1" />
                <INTEGER_CONSTANT value="10" type_id="integer" />
            </SUBSCRIPTS>
            <SUBSCRIPTS type_id="$anon_10e7b1b">
                <STRING_CONSTANT value="q" type_id="char" />
            </SUBSCRIPTS>
            <FIELD id="fldi" level="2" type_id="integer" />
        </VARIABLE>
    </ASSIGN>
    <IF line="57">
        <VARIABLE id="var9" level="1" type_id="boolean">
            <FIELD id="a" level="2" type_id="$anon_10c5dd24" />
            <SUBSCRIPTS type_id="boolean">
                <SUBTRACT type_id="integer">
                    <INTEGER_CONSTANT value="7" type_id="integer" />
                    <INTEGER_CONSTANT value="3" type_id="integer" />
                </SUBTRACT>
            </SUBSCRIPTS>
        </VARIABLE>
        <ASSIGN line="57" type_id="range2">
            <VARIABLE id="var2" level="1" type_id="range2">
                <SUBSCRIPTS type_id="range2">
                    <INTEGER_CONSTANT value="1" type_id="$anon_478eb1a9" />
                </SUBSCRIPTS>
            </VARIABLE>
```

```
                    <STRING_CONSTANT value="x" type_id="char" />
                </ASSIGN>
        </IF>
        <SELECT line="59">
            <VARIABLE id="var4" level="1" type_id="$anon_478eb1a9">
                <SUBSCRIPTS type_id="$anon_478eb1a9">
                    <SUBTRACT type_id="integer">
                        <VARIABLE id="var8" level="1" type_id="integer">
                            <SUBSCRIPTS type_id="$anon_10e7b1b">
                                <STRING_CONSTANT value="m" type_id="char" />
                            </SUBSCRIPTS>
                            <FIELD id="fldi" level="2" type_id="integer" />
                        </VARIABLE>
                        <INTEGER_CONSTANT value="4" type_id="integer" />
                    </SUBTRACT>
                </SUBSCRIPTS>
            </VARIABLE>
            <SELECT_BRANCH>
                <SELECT_CONSTANTS>
                    <INTEGER_CONSTANT value="0" type_id="$anon_478eb1a9" />
                </SELECT_CONSTANTS>
                <ASSIGN line="60" type_id="real">
                    <VARIABLE id="var3" level="1" type_id="real">
                        <SUBSCRIPTS type_id="arr1">
                            <INTEGER_CONSTANT value="4" type_id="enum1" />
                        </SUBSCRIPTS>
                        <SUBSCRIPTS type_id="real">
                            <INTEGER_CONSTANT value="3" type_id="integer" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                    <INTEGER_CONSTANT value="12" type_id="integer" />
                </ASSIGN>
            </SELECT_BRANCH>
            <SELECT_BRANCH>
                <SELECT_CONSTANTS>
                    <INTEGER_CONSTANT value="1" type_id="$anon_478eb1a9" />
                    <INTEGER_CONSTANT value="2" type_id="$anon_478eb1a9" />
                </SELECT_CONSTANTS>
                <ASSIGN line="61" type_id="arr1">
                    <VARIABLE id="var3" level="1" type_id="arr1">
                        <SUBSCRIPTS type_id="arr1">
                            <INTEGER_CONSTANT value="1" type_id="enum1" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                    <VARIABLE id="var1" level="1" type_id="arr1" />
                </ASSIGN>
            </SELECT_BRANCH>
        </SELECT>
        <LOOP line="64">
            <COMPOUND>
                <ASSIGN line="65" type_id="$anon_6097a37f">
                    <VARIABLE id="var5" level="1" type_id="$anon_6097a37f">
                        <SUBSCRIPTS type_id="$anon_6097a37f">
                            <INTEGER_CONSTANT value="3" type_id="integer" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                    <VARIABLE id="var5" level="1" type_id="$anon_6097a37f">
                        <SUBSCRIPTS type_id="$anon_6097a37f">
                            <INTEGER_CONSTANT value="4" type_id="integer" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                </ASSIGN>
                <ASSIGN line="66" type_id="$anon_5d0c0ae2">
                    <VARIABLE id="var5" level="1" type_id="$anon_5d0c0ae2">
                        <SUBSCRIPTS type_id="$anon_5d0c0ae2">
                            <INTEGER_CONSTANT value="3" type_id="integer" />
                            <STRING_CONSTANT value="a" type_id="char" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                    <VARIABLE id="var5" level="1" type_id="$anon_5d0c0ae2">
                        <SUBSCRIPTS type_id="$anon_6097a37f">
                            <INTEGER_CONSTANT value="4" type_id="integer" />
                        </SUBSCRIPTS>
                        <SUBSCRIPTS type_id="$anon_5d0c0ae2">
                            <STRING_CONSTANT value="f" type_id="char" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                </ASSIGN>
                <ASSIGN line="67" type_id="enum1">
                    <VARIABLE id="var5" level="1" type_id="enum1">
                        <SUBSCRIPTS type_id="enum1">
                            <INTEGER_CONSTANT value="3" type_id="integer" />
                            <STRING_CONSTANT value="a" type_id="char" />
                            <INTEGER_CONSTANT value="2" type_id="enum1" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                    <VARIABLE id="var6" level="1" type_id="enum1">
                        <SUBSCRIPTS type_id="$anon_5d0c0ae2">
                            <INTEGER_CONSTANT value="4" type_id="integer" />
                            <STRING_CONSTANT value="f" type_id="char" />
                        </SUBSCRIPTS>
                        <SUBSCRIPTS type_id="enum1">
                            <INTEGER_CONSTANT value="3" type_id="enum1" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                </ASSIGN>
            </COMPOUND>
            <TEST>
                <GT type_id="boolean">
                    <VARIABLE id="var6" level="1" type_id="enum1">
                        <SUBSCRIPTS type_id="enum1">
                            <INTEGER_CONSTANT value="3" type_id="integer" />
                            <STRING_CONSTANT value="a" type_id="char" />
                            <INTEGER_CONSTANT value="2" type_id="enum1" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                    <VARIABLE id="var5" level="1" type_id="enum1">
                        <SUBSCRIPTS type_id="$anon_6097a37f">
                            <INTEGER_CONSTANT value="4" type_id="integer" />
                        </SUBSCRIPTS>
                        <SUBSCRIPTS type_id="enum1">
                            <STRING_CONSTANT value="f" type_id="char" />
                            <INTEGER_CONSTANT value="3" type_id="enum1" />
                        </SUBSCRIPTS>
                    </VARIABLE>
                </GT>
            </TEST>
        </LOOP>
        <LOOP line="70">
            <TEST>
                <NOT>
                    <NE type_id="boolean">
```
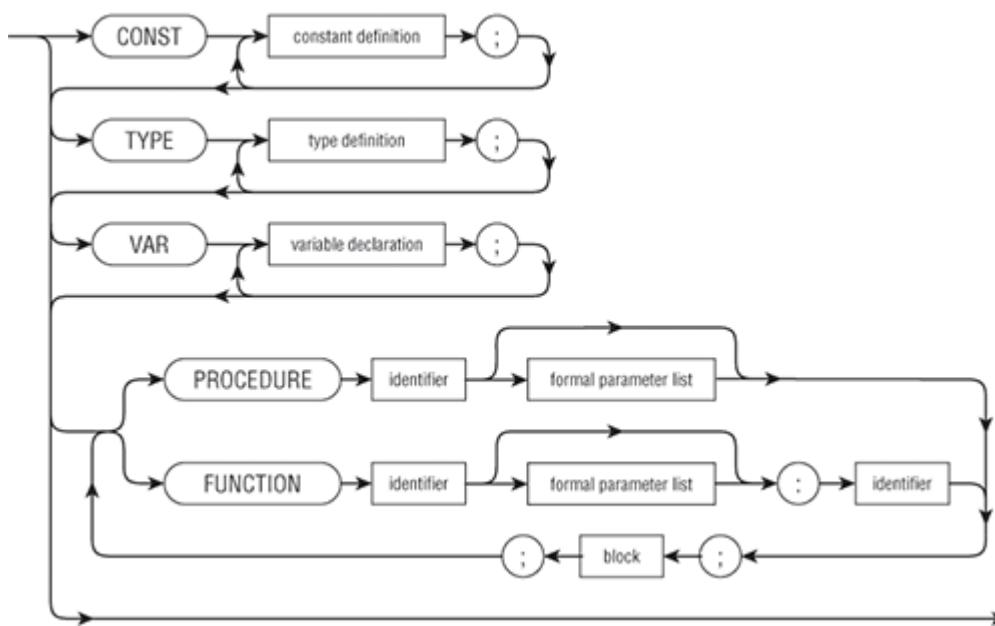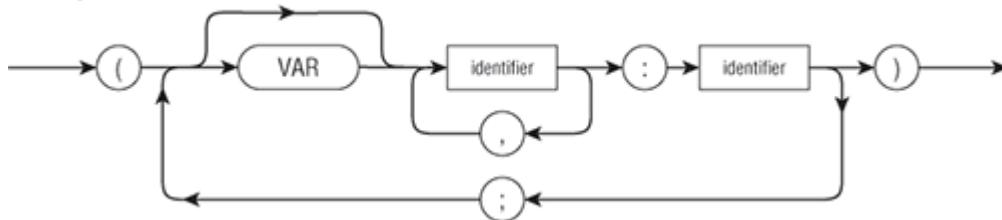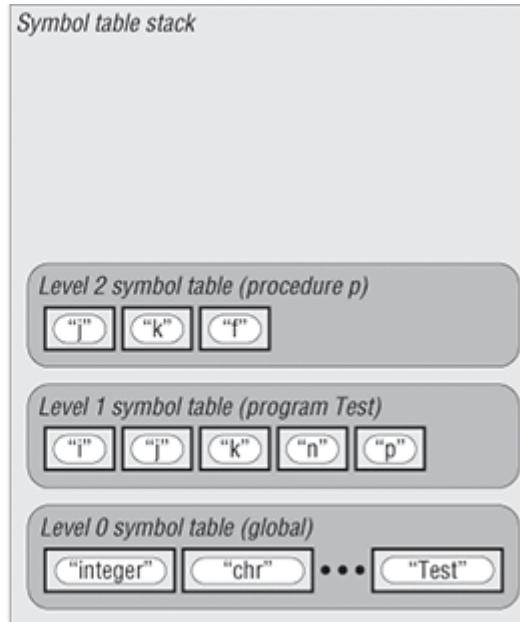
```
                <VARIABLE id="var7" level="1" type_id="integer">
                    <FIELD id="i" level="2" type_id="integer" />
                </VARIABLE>
                <VARIABLE id="var9" level="1" type_id="real">
                    <FIELD id="rec" level="2" type_id="$anon_16e7c77" />
                    <FIELD id="fldr" level="3" type_id="real" />
                </VARIABLE>
            </NE>
        </NOT>
    </TEST>

    <COMPOUND line="70">
        <ASSIGN line="71" type_id="real">
            <VARIABLE id="var7" level="1" type_id="real">
                <FIELD id="r" level="2" type_id="real" />
            </VARIABLE>
            <VARIABLE id="var8" level="1" type_id="real">
                <SUBSCRIPTS type_id="$anon_10e7b1b">
                    <STRING_CONSTANT value="g" type_id="char" />
                </SUBSCRIPTS>
                <FIELD id="fldr" level="2" type_id="rec7" />
                <FIELD id="r" level="2" type_id="real" />
            </VARIABLE>
        </ASSIGN>
    </COMPOUND>
</LOOP>
<ASSIGN line="74" type_id="$anon_6097a37f">
    <VARIABLE id="var6" level="1" type_id="$anon_6097a37f">
        <SUBSCRIPTS type_id="$anon_6097a37f">
            <INTEGER_CONSTANT value="3" type_id="integer" />
        </SUBSCRIPTS>
    </VARIABLE>
    <VARIABLE id="var6" level="1" type_id="$anon_6097a37f">
        <SUBSCRIPTS type_id="$anon_6097a37f">
            <INTEGER_CONSTANT value="4" type_id="integer" />
        </SUBSCRIPTS>
    </VARIABLE>
</ASSIGN>
<ASSIGN line="75" type_id="$anon_5d0c0ae2">
    <VARIABLE id="var6" level="1" type_id="$anon_5d0c0ae2">
        <SUBSCRIPTS type_id="$anon_5d0c0ae2">
            <INTEGER_CONSTANT value="3" type_id="integer" />
            <STRING_CONSTANT value="a" type_id="char" />
        </SUBSCRIPTS>
    </VARIABLE>
    <VARIABLE id="var6" level="1" type_id="$anon_5d0c0ae2">
        <SUBSCRIPTS type_id="$anon_6097a37f">
            <INTEGER_CONSTANT value="4" type_id="integer" />
        </SUBSCRIPTS>
        <SUBSCRIPTS type_id="$anon_5d0c0ae2">
            <STRING_CONSTANT value="f" type_id="char" />
        </SUBSCRIPTS>
    </VARIABLE>
</ASSIGN>
<ASSIGN line="77" type_id="enum1">
    <VARIABLE id="var9" level="1" type_id="enum1">
        <FIELD id="rec" level="2" type_id="$anon_16e7c77" />
        <FIELD id="fld6" level="3" type_id="arr6" />
        <SUBSCRIPTS type_id="$anon_6097a37f">
            <INTEGER_CONSTANT value="4" type_id="integer" />
        </SUBSCRIPTS>
        <SUBSCRIPTS type_id="$anon_5d0c0ae2">
            <STRING_CONSTANT value="f" type_id="char" />
        </SUBSCRIPTS>
        <SUBSCRIPTS type_id="enum1">
            <INTEGER_CONSTANT value="3" type_id="enum1" />
        </SUBSCRIPTS>
    </VARIABLE>
    <INTEGER_CONSTANT value="4" type_id="enum1" />
</ASSIGN>
<ASSIGN line="78" type_id="enum1">
    <VARIABLE id="var9" level="1" type_id="enum1">
        <FIELD id="rec" level="2" type_id="$anon_16e7c77" />
        <FIELD id="fld6" level="3" type_id="arr6" />
        <SUBSCRIPTS type_id="$anon_5d0c0ae2">
            <INTEGER_CONSTANT value="4" type_id="integer" />
            <STRING_CONSTANT value="f" type_id="char" />
        </SUBSCRIPTS>
        <SUBSCRIPTS type_id="enum1">
            <INTEGER_CONSTANT value="3" type_id="enum1" />
        </SUBSCRIPTS>
    </VARIABLE>
    <INTEGER_CONSTANT value="4" type_id="enum1" />
</ASSIGN>
<ASSIGN line="79" type_id="range2">
    <VARIABLE id="var9" level="1" type_id="range2">
        <FIELD id="rec" level="2" type_id="$anon_16e7c77" />
        <FIELD id="flda" level="3" type_id="$anon_b6ae9766" />
        <SUBSCRIPTS type_id="$anon_10e7b1b">
            <INTEGER_CONSTANT value="1" type_id="enum1" />
            <INTEGER_CONSTANT value="0" type_id="integer" />
            <STRING_CONSTANT value="m" type_id="char" />
        </SUBSCRIPTS>
        <FIELD id="flda" level="2" type_id="$anon_b3250579" />
        <SUBSCRIPTS type_id="range2">
            <INTEGER_CONSTANT value="3" type_id="enum1" />
        </SUBSCRIPTS>
    </VARIABLE>
    <STRING_CONSTANT value="p" type_id="char" />
</ASSIGN>
<ASSIGN line="80" type_id="range2">
    <VARIABLE id="var9" level="1" type_id="range2">
        <FIELD id="rec" level="2" type_id="$anon_16e7c77" />
        <FIELD id="flda" level="3" type_id="$anon_b6ae9766" />
        <SUBSCRIPTS type_id="$anon_8dbf888">
            <INTEGER_CONSTANT value="1" type_id="enum1" />
        </SUBSCRIPTS>
        <SUBSCRIPTS type_id="$anon_10e7b1b">
            <INTEGER_CONSTANT value="0" type_id="integer" />
            <STRING_CONSTANT value="m" type_id="char" />
        </SUBSCRIPTS>
        <FIELD id="flda" level="2" type_id="$anon_b3250579" />
        <SUBSCRIPTS type_id="range2">
            <INTEGER_CONSTANT value="3" type_id="enum1" />
        </SUBSCRIPTS>
    </VARIABLE>
    <STRING_CONSTANT value="p" type_id="char" />
</ASSIGN>
</COMPOUND>


        0 instructions generated.
        0.00 seconds total code generation time.
```

The printout of the parse tree now includes the type identifier for each node that has a type specification. Listing 10-22 shows a new version of method `printTypeSpec()` of class ParseTreePrinter in the util package. It generates an "artificial" type identifier for an unnamed type to make it easier to compare types visually.

Listing 10-22: Method `printTypeSpec()` of class `ParseTreePrinter`

```
/**
 * Print a parse tree node&apos;s type specification.
 * @param node the parse tree node.
 */
private void printTypeSpec(ICodeNodeImpl node)
{
    TypeSpec typeSpec = node.getTypeSpec();

    if (typeSpec != null) {
        String saveMargin = indentation;
        indentation += indent;

        String typeName;
        SymTabEntry typeId = typeSpec.getIdentifier();

        // Named type: Print the type identifier&apos;s name.
        if (typeId != null) {
            typeName = typeId.getName();
        }

        // Unnamed type: Print an artificial type identifier name.
        else {
            int code = typeSpec.hashCode() + typeSpec.getForm().hashCode();
            typeName = "$anon_" + Integer.toHexString(code);
        }

        printAttribute("TYPE_ID", typeName);
        indentation = saveMargin;
    }
}
```

Listing 10-23 shows output from the syntax checker when there are type-checking errors.

Listing 10-23: Output from the Pascal syntax checker when there are type-checking errors

```
001 CONST
002     seven =  7;
003     ten   = 10;
004
005 TYPE
006     range1 = 0..ten;
007     range2 = &apos;a&apos;..&apos;q&apos;;
008     range3 = range1;
009
010     enum1 = (a, b, c, d, e);
011     enum2 = enum1;
012
013     range4 = b..d;
014
015     arr1 = ARRAY [range1] OF real;
016     arr2 = ARRAY [(alpha, beta, gamma)] OF range2;
017     arr3 = ARRAY [enum2] OF arr1;
018     arr4 = ARRAY [range3] OF (foo, bar, baz);
019     arr5 = ARRAY [range1] OF ARRAY[range2] OF ARRAY[c..e] OF enum2;
020     arr6 = ARRAY [range1, range2, c..e] OF enum2;
021
022     rec7 = RECORD
023               i : integer;
024               r : real;
025               b1, b2 : boolean;
026               c : char
027           END;
028
029     arr8 = ARRAY [range2] OF RECORD
030                                 fldi  : integer;
031                                 fldr : rec7;
032                                 flda : ARRAY[range4] OF range2;
033                             END;
034
035 VAR
036     var1 : arr1;  var5 : arr5;
037     var2 : arr2;  var6 : arr6;
038     var3 : arr3;  var7 : rec7;
039     var4 : arr4;  var8 : arr8;
040
041     var9 : RECORD
042               b   : boolean;
043               rec : RECORD
044                        fld1 : arr1;
045                        fldb : boolean;
046                        fldr : real;
047                        fld6 : arr6;
048                        flda : ARRAY [enum1, range1] OF arr8;
049                     END;
050               a : ARRAY [1..5] OF boolean;
051           END;
052
053 BEGIN
054     var2[a] := 3.14;
              ^
*** Incompatible types [at "a"]
                 ^
*** Incompatible types [at "3.14"]
055     var1[var7.i] := var9.rec.flda[&apos;e&apos;, ten][&apos;q&apos;].fldr;
                                      ^
*** Incompatible types [at "&apos;e&apos;"]
                           ^
*** Incompatible types [at "var9"]
056
057     IF var9.rec.fldr THEN var2[beta] := seven;
           ^
*** Incompatible types [at "var9"]
                              ^
```

```
*** Incompatible types [at "seven"]

058
059    CASE var5[seven, &apos;m&apos;, d]  OF
060        foo:     var3[e] := 12;
                    ^
*** Incompatible types [at "foo"]

                                       ^
*** Incompatible types [at "12"]
061        bar, baz: var3[b] := var1.rec.fldb;
               ^
*** Incompatible types [at "bar"]

                  ^
*** Incompatible types [at "baz"]

                            ^
*** Invalid field [at "rec"]

                                ^
*** Invalid field [at "fldb"]
062    END;

063
064    REPEAT
065        var7[3] := a;
               ^
*** Too many subscripts [at "3"]

                   ^
*** Incompatible types [at "a"]
066    UNTIL var6[3, &apos;a&apos;, c] + var5[4][&apos;f&apos;, d];
                                 ^
*** Incompatible types [at "var5"]

              ^
*** Incompatible types [at "var6"]
067
068    var9.rec.flda[b][0, &apos;m&apos;, foo].flda[d] := &apos;p&apos;;
                              ^
*** Too many subscripts [at "foo"]
069 END.


             69 source lines.
             17 syntax errors.
           0.13 seconds total parsing time.
```

Design Note

By implementing type checking, you've removed Hack #4 described in Chapter 6.

In the next chapter, you'll complete the parsers to handle procedures, functions, and complete Pascal programs.

Chapter 11

Parsing Programs, Procedures, and Functions

In this chapter, you complete the parser that you've been working on since Chapter 5 in order to parse the Pascal program header, procedure and function declarations, and calls to procedures and functions.

Goals and Approach

The goal is to be able to parse complete Pascal programs, at least the subset of the source language that you've been using. As in the previous chapters, you'll develop parser subclasses to parse

- the Pascal program header
- Pascal procedure and function declarations
- calls to declared procedures and functions
- calls to the standard procedures and functions
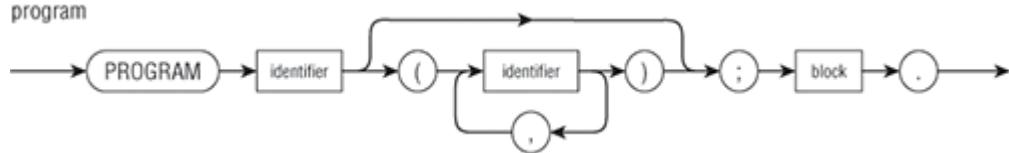
You'll make greater use of the symbol table stack. The latest edition of the ongoing test utility program, Syntax Checker IV, will verify your work at the end of the chapter.

Program, Procedure, and Function Declarations

You accomplished the bulk of parsing declarations in Chapter 9. Now complete the work by parsing a Pascal program declaration (the program header) and by parsing procedure and function declarations.

Figure 11-1 shows the syntax diagram for a complete Pascal program, including its program header. It refers to the diagram for a block from Figure 9-1.

Figure 11-1: Syntax diagrams for a Pascal program, including its program header

```
program

      PROGRAM      identifier      (      identifier      )      ;      block      .
                                                  ,
```

In the Pascal language specification, the header may optionally include a parenthesized comma-separated list of parameter identifiers. The identifiers represent files that are passed at runtime to the program (input files) or written by the program (output files). These parameters are optional if the program only reads from standard input and writes to standard output, which is all that your runtime executors will be able to do. Therefore, you'll parse the program parameters and verify their syntactic correctness, but otherwise ignore them.

Examples of program headers are:

```
PROGRAM WolfIsland;
PROGRAM XRef (input, output);
```

Standard input and standard output are by default the console if you run programs from the command line. You can use the < and > operators to redirect standard input and standard output, respectively, to text files. For example:

```
java -classpath classes Pascal execute wolfisland.pas < wolfisland.in
```

will compile and execute the Pascal program `wolfisland.pas`, and during run time, the program will read from the text file `wolfisland.pas`.

Figure 11-2 completes the syntax diagram in Figure 9-1 by including procedure and function declarations, each of which begins with the `PROCEDURE` or `FUNCTION` reserved word. Procedure and function declarations, if any, must come last in a block's declarations, and there may be any number of them in any order. The formal parameter list is optional, and a function declaration includes the type of its return value. Each procedure and function must have a block (or the `forward` identifier, as explained at the end of this section).

Figure 11-2: Syntax diagram for Pascal declarations, including procedures and functions

declarations

A declared procedure or function is a routine written by the programmer as part of the program. Standard procedures and functions are "built in" to Pascal and implicitly declared globally (at nesting level 0). Figure 11-3 shows the syntax diagram for the formal parameter list of a declared procedure or function.

Figure 11-3: Syntax diagram for a parameter list of a declared procedure or function

formal parameter list

Declared Pascal procedures and functions can be nested within each other to arbitrary depths. A function declaration includes a colon followed by the type identifier of the function return type. An unnamed type is not allowed.

The declaration of a formal parameter list should remind us of variable declarations diagrammed in Figure 9-1. A formal parameter list consists of one or more sublists separated by semicolons. Each sublist consists of one or more identifiers that are parameter names, and each sublist is followed by a colon and a type identifier that specifies the type of each parameter in the sublist. Unnamed types are not allowed. If a sublist is preceded by VAR, then each parameter in the sublist is a VAR parameter (i.e., passed by reference). Otherwise, each parameter in the sublist is passed by value.

Here are some examples of procedure and function declarations:

```
PROCEDURE proc (j, k : integer; VAR x, y, z : real; VAR v : arr;
               VAR p : boolean; ch : char);
   BEGIN
      ...
   END;

PROCEDURE SortWords;
   BEGIN
      ...
   END;

FUNCTION func (VAR x : real; i, n : integer) : real;
   BEGIN
      ...
      func := ...;
      ...
   END;
```

Note that procedure SortWords has no parameters.

Pascal does not have a return statement. A routine exits "naturally" when it reaches the end of its block. As shown above, sometime during the execution of its block, a function should assign its return value to the function identifier.

A Pascal program cannot call a procedure or a function that hasn't been declared yet. Therefore, it may be necessary to use a *forward declaration*, such as:

```
FUNCTION forwarded (m : integer; VAR t : real) : real; forward;
```

A forward declaration has the identifier forward instead of a block (forward is not a reserved word). Then later, the full declaration must appear:

```
FUNCTION forwarded;
   BEGIN
      ...
      forwarded := ...;
      ...
   END;
```
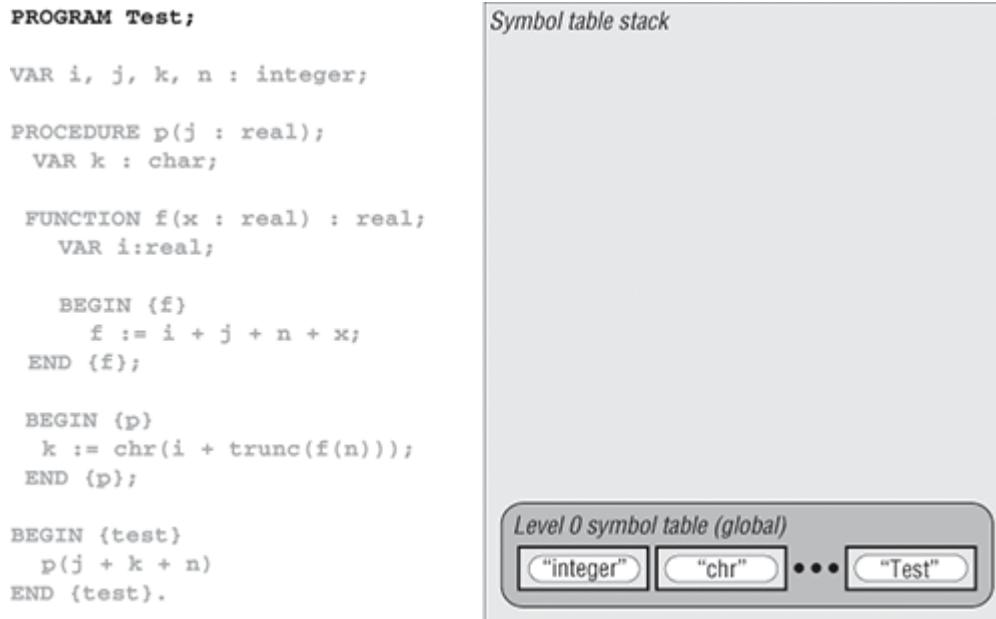
Note that the full declaration does not repeat the formal parameters, if any, or a function return type.

Nested Scopes and the Symbol Table Stack
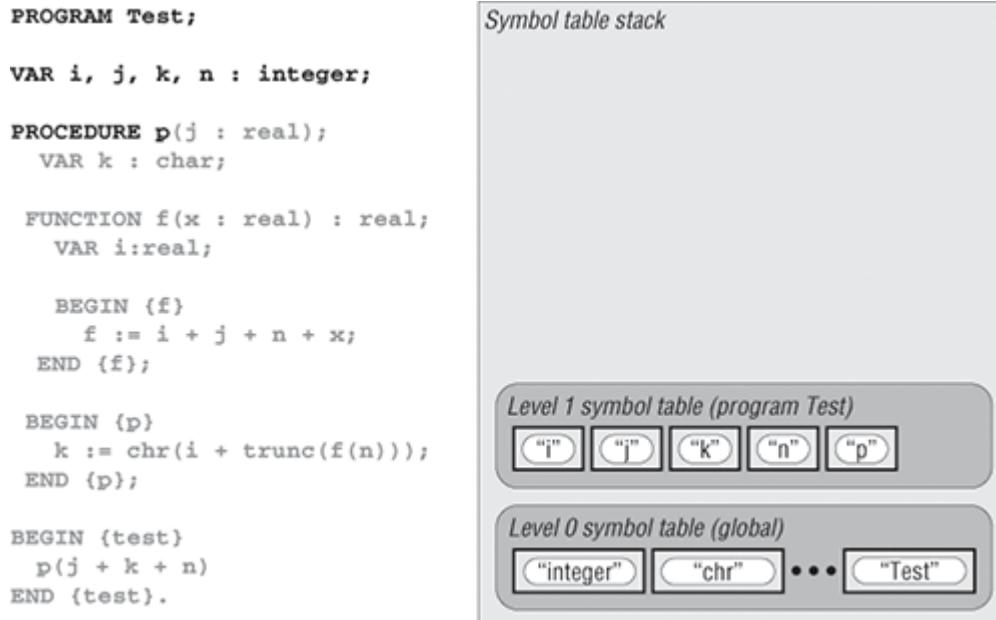
Because declared Pascal procedures and functions can be nested within each other, the symbol table stack plays a prominent role. Each procedure and function has its own symbol table to contain the identifiers that it defines. As the parser reads the source program from start to finish, it enters and exits scopes as it parses the routines. The parser pushes a new symbol table onto the stack for a routine when it begins to parse the routine, and later it pops the table off the stack when it's done parsing the routine.

Figures 11-4a through 11-4f show the various stages of the symbol table stack as the parser parses the contrived program `Test` with procedure `p` and nested function `f`.

Figure 11-4a: The symbol table stack after parsing the program header. The parser has entered name of the program `Test` into the Level 0 (global) symbol table.

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
 VAR k : char;

 FUNCTION f(x : real) : real;
   VAR i:real;

   BEGIN {f}
     f := i + j + n + x;
 END {f};

 BEGIN {p}
  k := chr(i + trunc(f(n)));
 END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```

Symbol table stack

Level 0 symbol table (global)
"integer"   "chr"   •••   "Test"

Figure 11-4b: The symbol table stack just after parsing the procedure name `p`. The parser pushed a new Level 1 (program) symbol table onto the stack and entered the main program's local variables `i`, `j`, `k`, and `n` along with procedure name `p` into the table.

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

 FUNCTION f(x : real) : real;
   VAR i:real;

   BEGIN {f}
     f := i + j + n + x;
  END {f};

 BEGIN {p}
   k := chr(i + trunc(f(n)));
 END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```

Symbol table stack

Level 1 symbol table (program Test)
"i" "j" "k" "n" "p"

Level 0 symbol table (global)
"integer"   "chr"   •••   "Test"

Figure 11-4c: The symbol table stack just after parsing the function name `f`. The parser pushed a new Level 2 symbol table onto the stack and entered the procedure's formal parameter `j` and local variable `k` along with function name `f` into the table.

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
  VAR i:real;

  BEGIN {f}
    f := i + j + n + x;
  END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```


Symbol table stack

Level 2 symbol table (procedure p) "j" "k" "f"

Level 1 symbol table (program Test) "i" "j" "k" "n" "p"

Level 0 symbol table (global) "integer" "chr" ••• "Test"

Figure 11-4d: The symbol table stack while parsing the body of function f. The parser pushed a new Level 3 symbol table onto the stack and entered the function's formal parameter x and local variable i into the table. When the parser searches the symbol table stack from top to bottom for the identifiers in the assignment statement, it finds x and i in the local table, j and f in the Level 2 table, and n in the Level 1 table.

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

  BEGIN {f}
    f := i + j + n + x;
  END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```
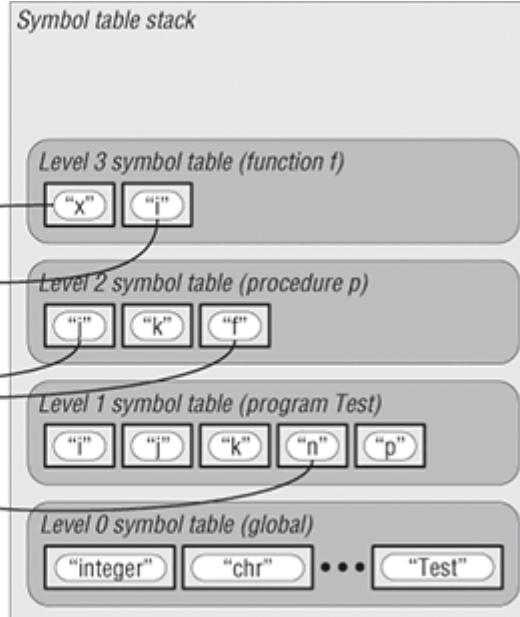

Symbol table stack

Level 3 symbol table (function f) "x" "i"

Level 2 symbol table (procedure p) "j" "k" "f"

Level 1 symbol table (program Test) "i" "j" "k" "n" "p"

Level 0 symbol table (global) "integer" "chr" ••• "Test"

Figure 11-4e: The symbol table stack while parsing the body of procedure p. The parser popped off the Level 3 symbol table. It finds identifiers k and f in the local table, i and n in the Level 1 table, and chr in the Level 0 table.

```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

  BEGIN {f}
    f := i + j + n + x;
  END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j + k + n)
END {test}.
```
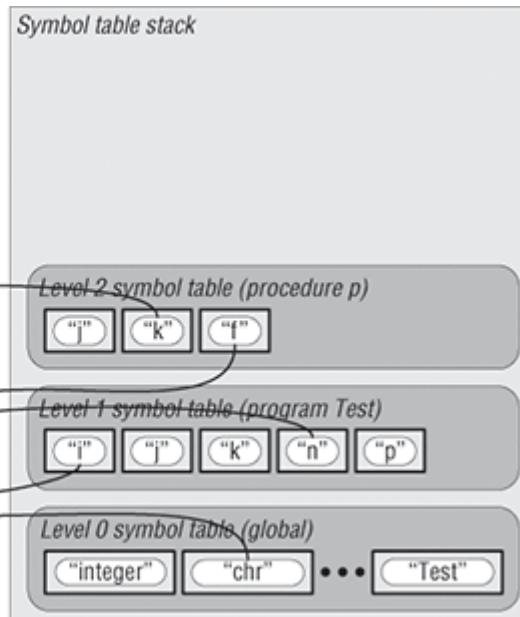

Symbol table stack

Level 2 symbol table (procedure p) "j" "k" "f"

Level 1 symbol table (program Test) "i" "j" "k" "n" "p"

Level 0 symbol table (global) "integer" "chr" ••• "Test"

Figure 11-4f: The symbol table stack while parsing the body of the main program. The parser popped off the Level 2 symbol table and it finds all the identifiers in the local table.
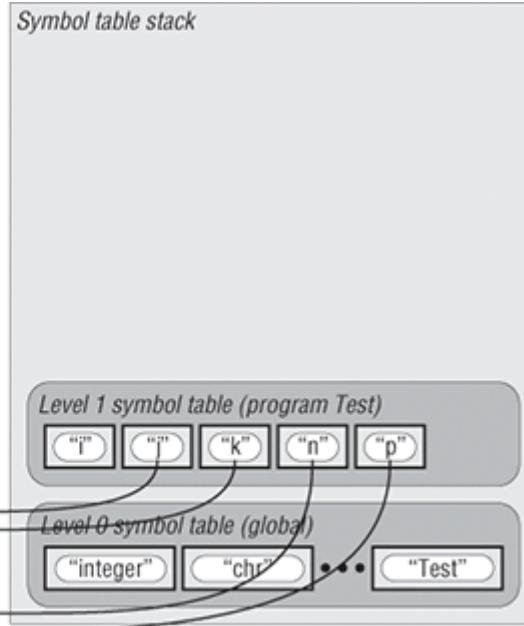
```
PROGRAM Test;

VAR i, j, k, n : integer;

PROCEDURE p(j : real);
  VAR k : char;

  FUNCTION f(x : real) : real;
    VAR i:real;

    BEGIN {f}
      f := i + j + n + x;
    END {f};

  BEGIN {p}
    k := chr(i + trunc(f(n)));
  END {p};

BEGIN {test}
  p(j) + (k) + (n)
END {test}.
```
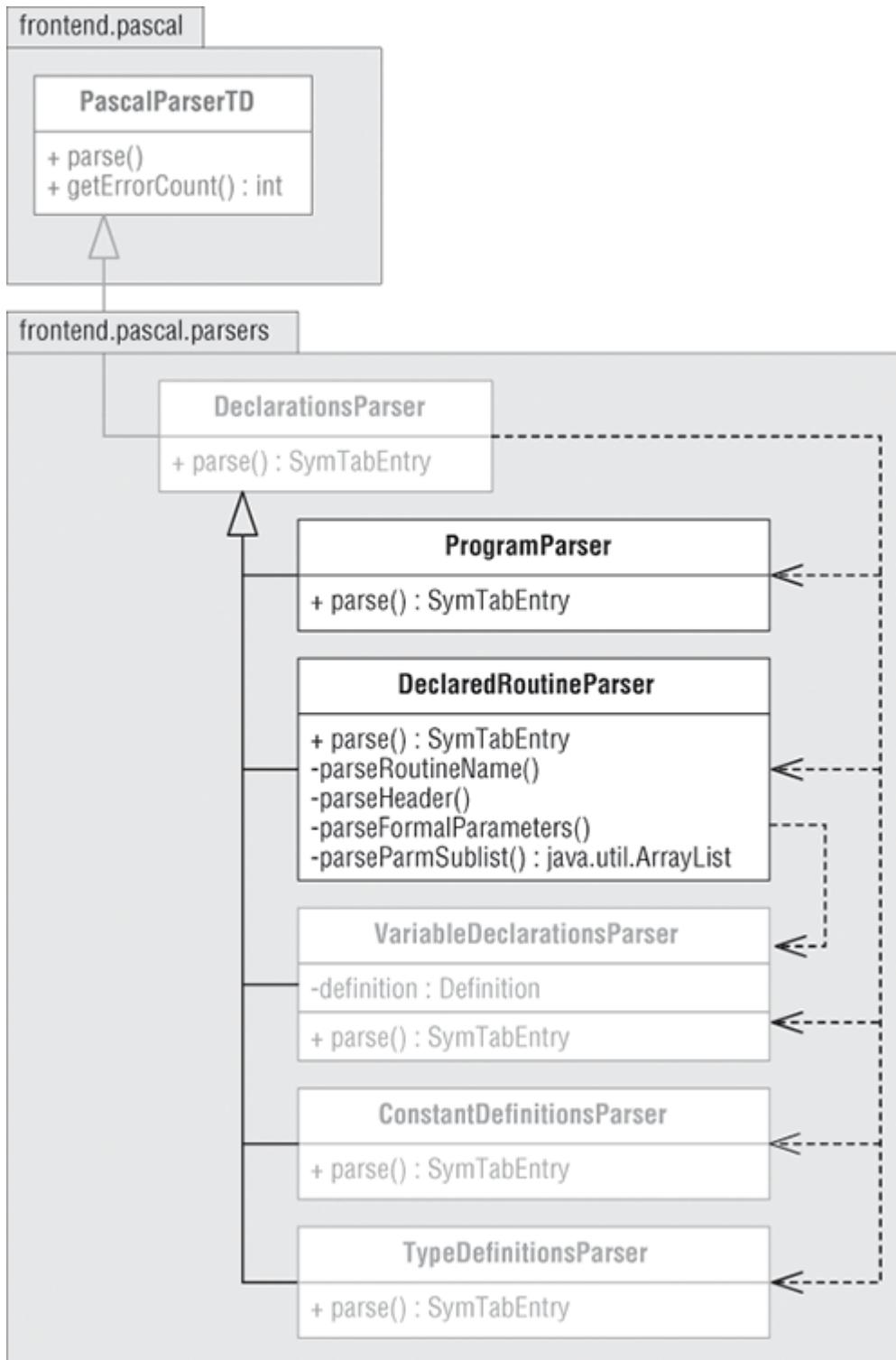

Note that the parser enters each routine's name into the symbol table of the enclosing routine. The parser enters function name f into the Level 2 symbol table of procedure p, procedure name p into the Level 1 symbol table of the main program, and the main program name Test into the global symbol table. This allows the assignment statement in the body of procedure p to refer to function f, as shown in Figure 11-4e.

New Declarations Parser Subclasses

Figure 11-5 extends the UML class diagram of Figure 9-5 to show the new declarations parser subclasses. Class ProgramParser parses the declaration of a program and class DeclaredRoutineParser parses declared procedures and functions. To accommodate parsing routines, method parse() of DeclarationsParser and its subclasses must return a symbol table entry. Each of these parse() methods returns null except for method DeclaredRoutineParser.parse() which returns the symbol table entry for the name of the routine it just parsed.

Listing 11-1 shows a new version of method parse() of class PascalParserTD. It calls programParser.parse() to parse an entire Pascal program.

Figure 11-5: The classes for parsing a Pascal program and declared procedures and functions.

frontend.pascal

PascalParserTD

+ parse()
+ getErrorCount() : int

frontend.pascal.parsers

DeclarationsParser

+ parse() : SymTabEntry

ProgramParser

+ parse() : SymTabEntry

DeclaredRoutineParser

+ parse() : SymTabEntry
-parseRoutineName()
-parseHeader()
-parseFormalParameters()
-parseParmSublist() : java.util.ArrayList

VariableDeclarationsParser

-definition : Definition

+ parse() : SymTabEntry

ConstantDefinitionsParser

+ parse() : SymTabEntry

TypeDefinitionsParser

+ parse() : SymTabEntry

Listing 11-1: Method `parse()` of class `PascalParserTD`

```
/**
 * Parse a Pascal source program and generate the symbol table
 * and the intermediate code.
 * @throws Exception if an error occurred.
 */
public void parse()
    throws Exception
{
    long startTime = System.currentTimeMillis();
    Predefined.initialize(symTabStack);

    try {
        Token token = nextToken();

        // Parse a program.
        ProgramParser programParser = new ProgramParser(this);
        programParser.parse(token, null);
        token = currentToken();

        // Send the parser summary message.
        float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
        sendMessage(new Message(PARSER_SUMMARY,
                            new Number[] {token.getLineNumber(),
                                          getErrorCount(),
                                          elapsedTime}));
    }
    catch (java.io.IOException ex) {
        errorHandler.abortTranslation(IO_ERROR, this);
    }
}
```

Now extend the `parse()` method of class `DeclarationsParser` to parse procedure and function declarations. See .

```
/**
 * Parse declarations.
 * To be overridden by the specialized declarations parser subclasses.
 * @param token the initial token.
 * @param parentId the symbol table entry of the parent routine&apos;s name.
 * @return null
 * @throws Exception if an error occurred.
 */
public SymTabEntry parse(Token token, SymTabEntry parentId)
    throws Exception
{
    token = synchronize(DECLARATION_START_SET);

    if (token.getType() == CONST) {
        token = nextToken();  // consume CONST

        ConstantDefinitionsParser constantDefinitionsParser =
            new ConstantDefinitionsParser(this);
        constantDefinitionsParser.parse(token, null);
    }

    token = synchronize(TYPE_START_SET);

    if (token.getType() == TYPE) {
        token = nextToken();  // consume TYPE

        TypeDefinitionsParser typeDefinitionsParser =
            new TypeDefinitionsParser(this);
        typeDefinitionsParser.parse(token, null);
    }

    token = synchronize(VAR_START_SET);

    if (token.getType() == VAR) {
        token = nextToken();  // consume VAR

        VariableDeclarationsParser variableDeclarationsParser =
            new VariableDeclarationsParser(this);
        variableDeclarationsParser.setDefinition(VARIABLE);
        variableDeclarationsParser.parse(token, null);
    }

    token = synchronize(ROUTINE_START_SET);
    TokenType tokenType = token.getType();

    while ((tokenType == PROCEDURE) || (tokenType == FUNCTION)) {
        DeclaredRoutineParser routineParser =
            new DeclaredRoutineParser(this);
        routineParser.parse(token, parentId);

        // Look for one or more semicolons after a definition.
        token = currentToken();
        if (token.getType() == SEMICOLON) {
            while (token.getType() == SEMICOLON) {
                token = nextToken();  // consume the ;
            }
        }

        token = synchronize(ROUTINE_START_SET);
        tokenType = token.getType();
    }

    return null;
}
```

Method `parse()` is passed the current token and the symbol table entry of the name of the routine that contains the routine about to be parsed. After it has seen and consumed the `PROCEDURE` or `FUNCTION` token, it calls `routineParser.parse()` to parse the rest of the routine. Then it looks for a semicolon and loops again to parse another procedure or function.

Parsing a Program Declaration

shows the `parse()` method of the declarations parser subclass `ProgramParser`. After the method has consumed the `PROGRAM` token, it calls `routineParser.parse()` to parse the rest of the program. Upon return, it looks for the final period token.

```
// Synchronization set to start a program.
static final EnumSet<PascalTokenType> PROGRAM_START_SET =
    EnumSet.of(PROGRAM, SEMICOLON);
static {
    PROGRAM_START_SET.addAll(DeclarationsParser.DECLARATION_START_SET);
}

/**
 * Parse a program.
 * @param token the initial token.
 * @param parentId the symbol table entry of the parent routine&apos;s name.
 * @return null
 * @throws Exception if an error occurred.
 */
public SymTabEntry parse(Token token, SymTabEntry parentId)
    throws Exception
{
    token = synchronize(PROGRAM_START_SET);

    // Parse the program.
    DeclaredRoutineParser routineParser = new DeclaredRoutineParser(this);
    routineParser.parse(token, parentId);

    // Look for the final period.
    token = currentToken();
    if (token.getType() != DOT) {
        errorHandler.flag(token, MISSING_PERIOD, this);
    }

    return null;
```

```
        }
```

Parsing Procedure and Function Declarations

The declarations parser subclass `DeclaredRoutineParser` does nearly all the work of parsing program, procedure, and function declarations.

shows its `parse()` method.

Listing 11-4: Method `parse()` of class `DeclaredRoutineParser`

```java
private static int dummyCounter = 0;  // counter for dummy routine names
private SymTabEntry parentId;         // entry of parent routine&apos;s name.

/**
 * Parse a standard subroutine declaration.
 * @param token the initial token.
 * @param parentId the symbol table entry of the parent routine&apos;s name.
 * @return the symbol table entry of the declared routine&apos;s name.
 * @throws Exception if an error occurred.
 */
public SymTabEntry parse(Token token, SymTabEntry parentId)
    throws Exception
{
    Definition routineDefn = null;
    String dummyName = null;
    SymTabEntry routineId = null;
    TokenType routineType = token.getType();

    // Initialize.
    switch ((PascalTokenType) routineType) {

        case PROGRAM: {
            token = nextToken();  // consume PROGRAM
            routineDefn = DefinitionImpl.PROGRAM;
            dummyName = "DummyProgramName".toLowerCase();
            break;
        }

        case PROCEDURE: {
            token = nextToken();  // consume PROCEDURE
            routineDefn = DefinitionImpl.PROCEDURE;
            dummyName = "DummyProcedureName_".toLowerCase() +
                        String.format("%03d", ++dummyCounter);
            break;
        }

        case FUNCTION: {
            token = nextToken();  // consume FUNCTION
            routineDefn = DefinitionImpl.FUNCTION;
            dummyName = "DummyFunctionName_".toLowerCase() +
                        String.format("%03d", ++dummyCounter);
            break;
        }

        default: {
            routineDefn = DefinitionImpl.PROGRAM;
            dummyName = "DummyProgramName".toLowerCase();
            break;
        }
    }

    // Parse the routine name.
    routineId = parseRoutineName(token, dummyName);
    routineId.setDefinition(routineDefn);

    token = currentToken();

    // Create new intermediate code for the routine.
    ICode iCode = ICodeFactory.createICode();
    routineId.setAttribute(ROUTINE_ICODE, iCode);
    routineId.setAttribute(ROUTINE_ROUTINES, new ArrayList<SymTabEntry>());

    // Push the routine&apos;s new symbol table onto the stack.
    // If it was forwarded, push its existing symbol table.
    if (routineId.getAttribute(ROUTINE_CODE) == FORWARD) {
        SymTab symTab = (SymTab) routineId.getAttribute(ROUTINE_SYMTAB);
        symTabStack.push(symTab);
    }
    else {
        routineId.setAttribute(ROUTINE_SYMTAB, symTabStack.push());
    }

    // Program: Set the program identifier in the symbol table stack.
    if (routineDefn == DefinitionImpl.PROGRAM) {
        symTabStack.setProgramId(routineId);
    }

    // Non-forwarded procedure or function: Append to the parent&apos;s list
    //                                      of routines.
    else if (routineId.getAttribute(ROUTINE_CODE) != FORWARD) {
        ArrayList<SymTabEntry> subroutines = (ArrayList<SymTabEntry>)
                            parentId.getAttribute(ROUTINE_ROUTINES);
        subroutines.add(routineId);
    }

    // If the routine was forwarded, there should not be
    // any formal parameters or a function return type.
    // But parse them anyway if they&apos;re there.
    if (routineId.getAttribute(ROUTINE_CODE) == FORWARD) {
        if (token.getType() != SEMICOLON) {
            errorHandler.flag(token, ALREADY_FORWARDED, this);
            parseHeader(token, routineId);
        }
    }

    // Parse the routine&apos;s formal parameters and function return type.
    else {
        parseHeader(token, routineId);
    }

    // Look for the semicolon.
    token = currentToken();
    if (token.getType() == SEMICOLON) {
        do {
```

```
            token = nextToken();  // consume ;
        } while (token.getType() == SEMICOLON);
    }
    else {
        errorHandler.flag(token, MISSING_SEMICOLON, this);
    }

    // Parse the routine&apos;s block or forward declaration.
    if ((token.getType() == IDENTIFIER) &&
        (token.getText().equalsIgnoreCase("forward")))
    {
        token = nextToken();  // consume forward
        routineId.setAttribute(ROUTINE_CODE, FORWARD);
    }
    else {
        routineId.setAttribute(ROUTINE_CODE, DECLARED);

        BlockParser blockParser = new BlockParser(this);
        ICodeNode rootNode = blockParser.parse(token, routineId);
        iCode.setRoot(rootNode);
    }

    // Pop the routine&apos;s symbol table off the stack.
    symTabStack.pop();

    return routineId;
}
```

After consuming the `PROGRAM`, `PROCEDURE`, or `FUNCTION` token and choosing the appropriate identifier definition, method `parse()` calls `parseRoutineName()` to parse the routine name. It creates new intermediate code for the routine and stores it as the `ROUTINE_ICODE` attribute of the routine name's symbol table entry. It pushes a new symbol table onto the symbol table stack and also stores the table as the `ROUTINE_SYMTAB` attribute of the routine names' symbol table entry. But if the procedure or function was previously declared in a forward declaration, the method instead pushes the symbol table that was created while parsing the forward declaration.

If there was a forward declaration, there shouldn't be a redeclaration of the formal parameters or the function return type, but method `parse()` will parse them anyway for error recovery. In any case, it calls `parseHeader()` to parse the parameters and the function return type.

After parsing the parameters and the return type, method `parse()` either sees and consumes the identifier `forward`, or it calls `blockParser.parse()` to parse the routine's block. When it's done parsing the routine, the method pops the routine's symbol table off the symbol table stack and returns the routine name's symbol table entry.

Design Note

After the parser finishes parsing a routine and pops the routine's symbol table off the stack, the table doesn't disappear. As shown in Listing 11-4, the `ROUTINE_SYMTAB` attribute of the routine name's symbol table entry maintains a reference to the table. Similarly, the symbol table entry's `ROUTINE_ICODE` attribute maintains a reference to the parse tree generated for the routine's block. The back end interpreter or compiler later will need to access each routine's symbol table and parse tree.

Listing 11-5 shows methods `parseRoutineName()` and `parseHeader()`.

Listing 11-5: **Methods `parseRoutineName()` and `parseHeader()` of class `DeclaredRoutineParser`**

```
/**
 * Parse a routine&apos;s name.
 * @param token the current token.
 * @param routineDefn how the routine is defined.
 * @param dummyName a dummy name in case of parsing problem.
 * @throws Exception if an error occurred.
 * @return the symbol table entry of the declared routine&apos;s name.
 */
private SymTabEntry parseRoutineName(Token token, String dummyName)
    throws Exception
{
    SymTabEntry routineId = null;

    // Parse the routine name identifier.
    if (token.getType() == IDENTIFIER) {
        String routineName = token.getText().toLowerCase();
        routineId = symTabStack.lookupLocal(routineName);

        // Not already defined locally: Enter into the local symbol table.
        if (routineId == null) {
            routineId = symTabStack.enterLocal(routineName);
        }

        // If already defined, it should be a forward definition.
        else if (routineId.getAttribute(ROUTINE_CODE) != FORWARD) {
            routineId = null;
            errorHandler.flag(token, IDENTIFIER_REDEFINED, this);
        }

        token = nextToken();  // consume routine name identifier
    }
    else {
        errorHandler.flag(token, MISSING_IDENTIFIER, this);
    }

    // If necessary, create a dummy routine name symbol table entry.
    if (routineId == null) {
        routineId = symTabStack.enterLocal(dummyName);
    }

    return routineId;
}

/**
 * Parse a routine&apos;s formal parameter list and the function return type.
 * @param token the current token.
 * @param routineId the symbol table entry of the declared routine&apos;s name.
 * @throws Exception if an error occurred.
 */
private void parseHeader(Token token, SymTabEntry routineId)
    throws Exception
{
```

```
        // Parse the routine&apos;s formal parameters.
        parseFormalParameters(token, routineId);
        token = currentToken();

        // If this is a function, parse and set its return type.
        if (routineId.getDefinition() == DefinitionImpl.FUNCTION) {
            VariableDeclarationsParser variableDeclarationsParser =
                new VariableDeclarationsParser(this);
            variableDeclarationsParser.setDefinition(DefinitionImpl.FUNCTION);
            TypeSpec type = variableDeclarationsParser.parseTypeSpec(token);

            token = currentToken();

            // The return type cannot be an array or record.
            if (type != null) {
                TypeForm form = type.getForm();
                if ((form == TypeFormImpl.ARRAY) ||
                    (form == TypeFormImpl.RECORD))
                {
                    errorHandler.flag(token, INVALID_TYPE, this);
                }
            }

            // Missing return type.
            else {
                type = Predefined.undefinedType;
            }

            routineId.setTypeSpec(type);
            token = currentToken();
        }
    }
```

Method `parseRoutineName()` verifies that if the procedure or function name is already in the local symbol table, it must have come from a previous forward declaration.

Method `parseHeader()` calls `parseFormalParameters()` to parse the routine's formal parameter list. If the routine is a function, it also calls `variableDeclarationsParser.parseTypeSpec()` to parse the return type. Pascal does not allow a function's return type to be an array or a record.

Formal Parameter Lists

Listing 11-6 shows method `parseFormalParameters()`.

Listing 11-6: Method `parseFormalParameters()` of class `DeclaredRoutineParser`

```
    // Synchronization set for a formal parameter sublist.
    private static final EnumSet<PascalTokenType> PARAMETER_SET =
        DeclarationsParser.DECLARATION_START_SET.clone();
    static {
        PARAMETER_SET.add(VAR);
        PARAMETER_SET.add(IDENTIFIER);
        PARAMETER_SET.add(RIGHT_PAREN);
    }

    // Synchronization set for the opening left parenthesis.
    private static final EnumSet<PascalTokenType> LEFT_PAREN_SET =
        DeclarationsParser.DECLARATION_START_SET.clone();
    static {
        LEFT_PAREN_SET.add(LEFT_PAREN);
        LEFT_PAREN_SET.add(SEMICOLON);
        LEFT_PAREN_SET.add(COLON);
    }

    // Synchronization set for the closing right parenthesis.
    private static final EnumSet<PascalTokenType> RIGHT_PAREN_SET =
        LEFT_PAREN_SET.clone();
    static {
        RIGHT_PAREN_SET.remove(LEFT_PAREN);
        RIGHT_PAREN_SET.add(RIGHT_PAREN);
    }

    /**
     * Parse a routine&apos;s formal parameter list.
     * @param token the current token.
     * @param routineId the symbol table entry of the declared routine&apos;s name.
     * @throws Exception if an error occurred.
     */
    protected void parseFormalParameters(Token token, SymTabEntry routineId)
        throws Exception
    {
        // Parse the formal parameters if there is an opening left parenthesis.
        token = synchronize(LEFT_PAREN_SET);
        if (token.getType() == LEFT_PAREN) {
            token = nextToken();  // consume (

            ArrayList<SymTabEntry> parms = new ArrayList<SymTabEntry>();

            token = synchronize(PARAMETER_SET);
            TokenType tokenType = token.getType();

            // Loop to parse sublists of formal parameter declarations.
            while ((tokenType == IDENTIFIER) || (tokenType == VAR)) {
                parms.addAll(parseParmSublist(token, routineId));
                token = currentToken();
                tokenType = token.getType();
            }

            // Closing right parenthesis.
            if (token.getType() == RIGHT_PAREN) {
                token = nextToken();  // consume )
            }
            else {
                errorHandler.flag(token, MISSING_RIGHT_PAREN, this);
            }

            routineId.setAttribute(ROUTINE_PARMS, parms);
        }
    }
```

If there are formal parameters, method `parseFormalParameters()` loops to call `parseParmSublist()` to parse each sublist of parameter declarations. It builds a list of symbol table entries of the parameter identifiers which it sets as the value of the `ROUTINE_PARMS` attribute of the routine name's symbol table entry.

shows method `parseParmSublist()`.

Listing 11-7: Method `parseParmSublist()` of class `DeclaredRoutineParser`

```
    // Synchronization set to follow a formal parameter identifier.
    private static final EnumSet<PascalTokenType> PARAMETER_FOLLOW_SET =
        EnumSet.of(COLON, RIGHT_PAREN, SEMICOLON);
    static {
        PARAMETER_FOLLOW_SET.addAll(DeclarationsParser.DECLARATION_START_SET);
    }

    // Synchronization set for the , token.
    private static final EnumSet<PascalTokenType> COMMA_SET =
        EnumSet.of(COMMA, COLON, IDENTIFIER, RIGHT_PAREN, SEMICOLON);
    static {
        COMMA_SET.addAll(DeclarationsParser.DECLARATION_START_SET);
    }

    /**
     * Parse a sublist of formal parameter declarations.
     * @param token the current token.
     * @param routineId the symbol table entry of the declared routine&apos;s name.
     * @return the sublist of symbol table entries for the parm identifiers.
     * @throws Exception if an error occurred.
     */
    private ArrayList<SymTabEntry> parseParmSublist(Token token,
                                                    SymTabEntry routineId)
        throws Exception
    {
        boolean isProgram = routineId.getDefinition() == DefinitionImpl.PROGRAM;
        Definition parmDefn = isProgram ? PROGRAM_PARM : null;
        TokenType tokenType = token.getType();

        // VAR or value parameter?
        if (tokenType == VAR) {
            if (!isProgram) {
                parmDefn = VAR_PARM;
            }
            else {
                errorHandler.flag(token, INVALID_VAR_PARM, this);
            }

            token = nextToken();  // consume VAR
        }
        else if (!isProgram) {
            parmDefn = VALUE_PARM;
        }

        // Parse the parameter sublist and its type specification.
        VariableDeclarationsParser variableDeclarationsParser =
            new VariableDeclarationsParser(this);
        variableDeclarationsParser.setDefinition(parmDefn);
        ArrayList<SymTabEntry> sublist =
            variableDeclarationsParser.parseIdentifierSublist(
                                         token, PARAMETER_FOLLOW_SET,
                                         COMMA_SET);
        token = currentToken();
        tokenType = token.getType();

        if (!isProgram) {

            // Look for one or more semicolons after a sublist.
            if (tokenType == SEMICOLON) {
                while (token.getType() == SEMICOLON) {
                    token = nextToken();  // consume the ;
                }
            }

            // If at the start of the next sublist, then missing a semicolon.
            else if (VariableDeclarationsParser.
                        NEXT_START_SET.contains(tokenType)) {
                errorHandler.flag(token, MISSING_SEMICOLON, this);
            }

            token = synchronize(PARAMETER_SET);
        }

        return sublist;
    }
```

In method `parseParmSublist()`, a VAR token means that each formal parameter in the sublist is defined to be a VAR_PARM (passed by reference). Otherwise, each parameter is defined to be a VALUE_PARM. However, program parameters can only be defined to be PROGRAM_PARM. The method calls `variableDeclarationsParser.setDefinition()` to let that parser know how to define the parameters, and then it calls `variableDeclarationsParser.parseIdentifierSublist()` to do the work.

Finally, shows a new version of method `parseIdentifierSublist()` of the declarations subclass `VariableDeclarationsParser`.

Listing 11-8: Method `parseIdentifierSublist()` of class `VariableDeclarationsParser`

```
    /**
     * Parse a sublist of identifiers and their type specification.
     * @param token the current token.
     * @param followSet the synchronization set to follow an identifier.
     * @return the sublist of identifiers in a declaration.
     * @throws Exception if an error occurred.
     */
    protected ArrayList<SymTabEntry> parseIdentifierSublist(
                                     Token token,
                                     EnumSet<PascalTokenType> followSet,
                                     EnumSet<PascalTokenType> commaSet)
        throws Exception
    {
        ArrayList<SymTabEntry> sublist = new ArrayList<SymTabEntry>();

        do {
            token = synchronize(IDENTIFIER_START_SET);
            SymTabEntry id = parseIdentifier(token);

            if (id != null) {
                sublist.add(id);
            }

            token = synchronize(commaSet);
            TokenType tokenType = token.getType();
```

```
            // Look for the comma.
            if (tokenType == COMMA) {
                token = nextToken();  // consume the comma

                if (followSet.contains(token.getType())) {
                    errorHandler.flag(token, MISSING_IDENTIFIER, this);
                }
            }
            else if (IDENTIFIER_START_SET.contains(tokenType)) {
                errorHandler.flag(token, MISSING_COMMA, this);
            }
        } while (!followSet.contains(token.getType()));

        if (definition != PROGRAM_PARM) {

            // Parse the type specification.
            TypeSpec type = parseTypeSpec(token);

            // Assign the type specification to each identifier in the list.
            for (SymTabEntry variableId : sublist) {
                variableId.setTypeSpec(type);
            }
        }
    }
    return sublist;
}
```

In order to accommodate parsing sublists of formal parameter declarations and not just sublists of variable declarations, expand method `parseIdentifierSublist()` by changing its signature and making it more flexible. The method must not parse and assign a type specification for program parameters.

Method `parse()` of class `VariableDeclarationsParser` then must call `parseIdentifierSublist()` with two additional arguments:

```
// Parse the identifier sublist and its type specification.
parseIdentifierSublist(token, IDENTIFIER_FOLLOW_SET, COMMA_SET);
```

You must also modify the `parse()` method of class `BlockParser` to accommodate parsing routines. See Listing 11-9.

Listing 11-9: Method `parse()` of class `BlockParser`

```
/**
 * Parse a block.
 * @param token the initial token.
 * @param routineId the symbol table entry of the routine name.
 * @return the root node of the parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token, SymTabEntry routineId)
    throws Exception
{
    DeclarationsParser declarationsParser = new DeclarationsParser(this);
    StatementParser statementParser = new StatementParser(this);

    // Parse any declarations.
    declarationsParser.parse(token, routineId);

    token = synchronize(StatementParser.STMT_START_SET);
    TokenType tokenType = token.getType();
    ICodeNode rootNode = null;

    // Look for the BEGIN token to parse a compound statement.
    if (tokenType == BEGIN) {
        rootNode = statementParser.parse(token);
    }

    // Missing BEGIN: Attempt to parse anyway if possible.
    else {
        errorHandler.flag(token, MISSING_BEGIN, this);

        if (StatementParser.STMT_START_SET.contains(tokenType)) {
            rootNode = ICodeFactory.createICodeNode(COMPOUND);
            statementParser.parseList(token, rootNode, END, MISSING_END);
        }
    }

    return rootNode;
}
```
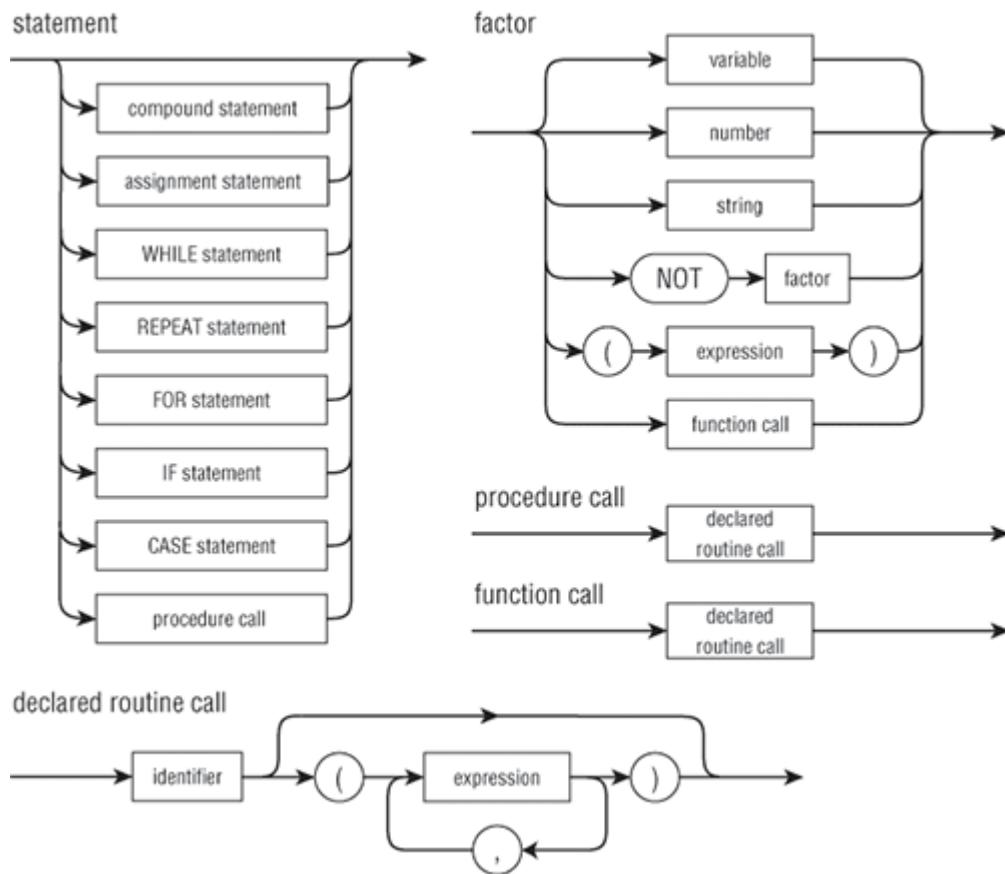
This version of method `parse()` is passed the symbol table entry of the parent routine's name, which the method in turn passes in its call to `declarationsParser.parse()`.

Parsing Procedure and Function Calls

Figure 11-6 shows the syntax diagrams of a procedure call as a statement and a function call as a factor. These diagrams extend the syntax diagrams of a factor in Figure 5-2 and a statement in Figure 7-1.
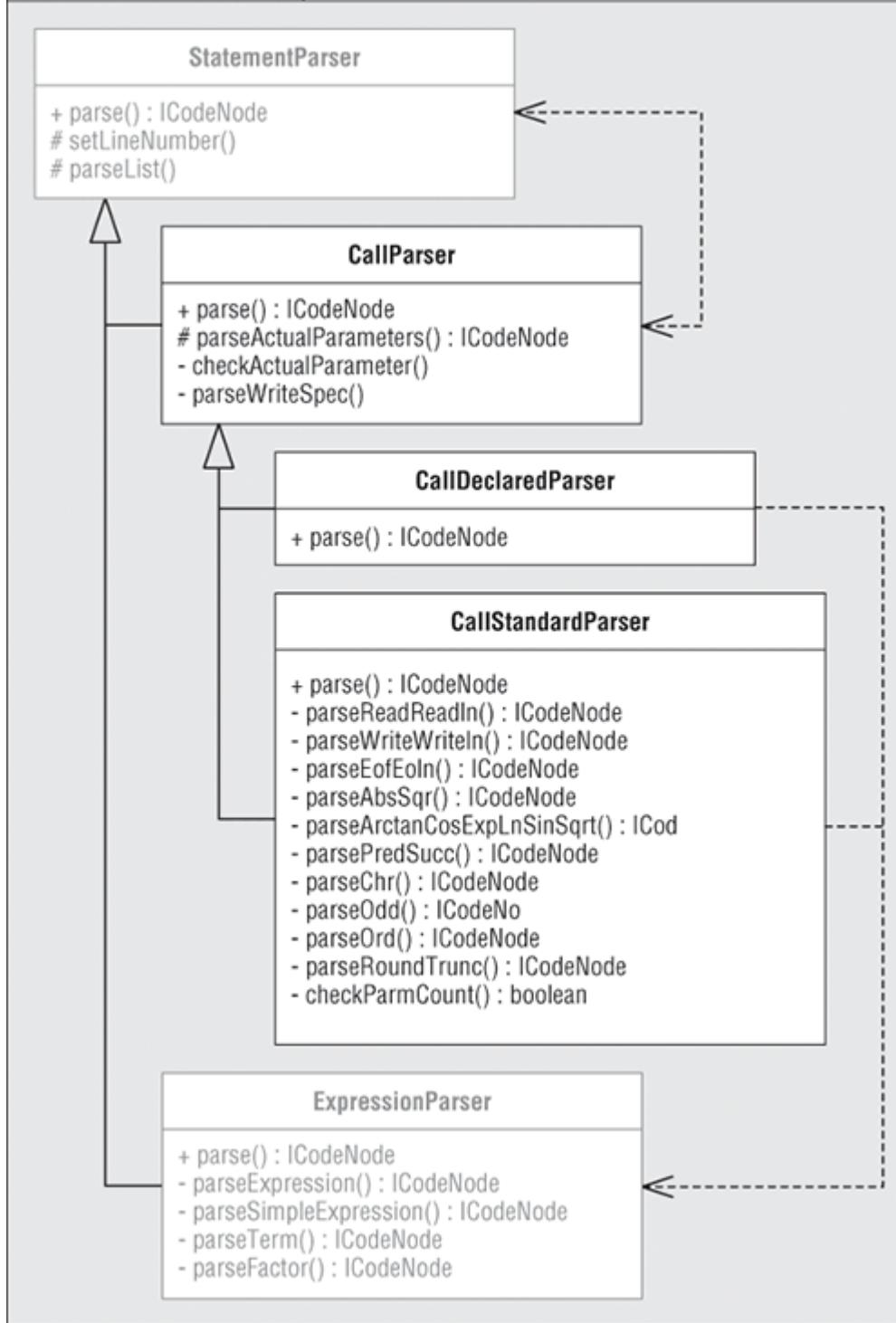
Figure 11-6: Syntax diagrams for a procedure call as a statement and a function call as a factor

Figure 11-7 shows the UML class diagram for the statement parser subclasses that will parse calls to declared procedures and functions and to the standard procedures and functions.

The statement parser subclass `CallParser` has two subclasses, `CallDeclaredParser` and `CallStandardParser`. Both subclasses use its `parseActualParameters()` method.

Figure 11-7: The classes for parsing calls to declared and the standard procedures and functions

```
frontend.pascal.parsers

┌─────────────────────────────────────┐
│           StatementParser            │
├─────────────────────────────────────┤
│ + parse() : ICodeNode                │
│ # setLineNumber()                    │
│ # parseList()                        │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│              CallParser              │
├─────────────────────────────────────┤
│ + parse() : ICodeNode                │
│ # parseActualParameters() : ICodeNode│
│ - checkActualParameter()             │
│ - parseWriteSpec()                   │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│          CallDeclaredParser          │
├─────────────────────────────────────┤
│ + parse() : ICodeNode                │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│          CallStandardParser          │
├─────────────────────────────────────┤
│ + parse() : ICodeNode                │
│ - parseReadReadln() : ICodeNode      │
│ - parseWriteWriteln() : ICodeNode    │
│ - parseEofEoln() : ICodeNode         │
│ - parseAbsSqr() : ICodeNode          │
│ - parseArctanCosExpLnSinSqrt() : ICod│
│ - parsePredSucc() : ICodeNode        │
│ - parseChr() : ICodeNode             │
│ - parseOdd() : ICodeNo               │
│ - parseOrd() : ICodeNode             │
│ - parseRoundTrunc() : ICodeNode      │
│ - checkParmCount() : boolean         │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│           ExpressionParser           │
├─────────────────────────────────────┤
│ + parse() : ICodeNode                │
│ - parseExpression() : ICodeNode      │
│ - parseSimpleExpression() : ICodeNode│
│ - parseTerm() : ICodeNode            │
│ - parseFactor() : ICodeNode          │
└─────────────────────────────────────┘
```

You need separate parsers for calls to declared routines and for calls to Pascal's standard routines. The standard routines are somewhat *non*standard in the way calls are made to them! For example, procedure `read` has a variable number of actual parameters. Procedures `write` and `writeln` can have optional colon-separated field width and precision numbers after each parameter. Functions such as `abs` can have either an integer or real parameter and return a value of the same type as the parameter. For these reasons, class `CallStandardParser` will have *ad hoc* methods for parsing the calls.

Listing 11-10 shows additions to class `Predefined` in package `intermediate.symtabimpl` to enter the names of Pascal's standard procedures and functions into the global symbol table.

Listing 11-10: Additions to class `Predefined`

```
public static SymTabEntry readId;
public static SymTabEntry readlnId;
public static SymTabEntry writeId;
public static SymTabEntry writelnId;
public static SymTabEntry absId;
public static SymTabEntry arctanId;
public static SymTabEntry chrId;
public static SymTabEntry cosId;
public static SymTabEntry eofId;
public static SymTabEntry eolnId;
public static SymTabEntry expId;
public static SymTabEntry lnId;
public static SymTabEntry oddId;
public static SymTabEntry ordId;
public static SymTabEntry predId;
public static SymTabEntry roundId;
```

```java
    public static SymTabEntry sinId;
    public static SymTabEntry sqrId;
    public static SymTabEntry sqrtId;
    public static SymTabEntry succId;
    public static SymTabEntry truncId;

    /**
     * Initialize a symbol table stack with predefined identifiers.
     * @param symTab the symbol table stack to initialize.
     */
    public static void initialize(SymTabStack symTabStack)
    {
        initializeTypes(symTabStack);
        initializeConstants(symTabStack);
        initializeStandardRoutines(symTabStack);
    }


    /**
     * Initialize the standard procedures and functions.
     * @param symTabStack the symbol table stack to initialize.
     */
    private static void initializeStandardRoutines(SymTabStack symTabStack)
    {
        readId    = enterStandard(symTabStack, PROCEDURE, "read",    READ);
        readlnId  = enterStandard(symTabStack, PROCEDURE, "readln",  READLN);
        writeId   = enterStandard(symTabStack, PROCEDURE, "write",   WRITE);
        writelnId = enterStandard(symTabStack, PROCEDURE, "writeln", WRITELN);

        absId    = enterStandard(symTabStack, FUNCTION, "abs",    ABS);
        arctanId = enterStandard(symTabStack, FUNCTION, "arctan", ARCTAN);
        chrId    = enterStandard(symTabStack, FUNCTION, "chr",    CHR);
        cosId    = enterStandard(symTabStack, FUNCTION, "cos",    COS);
        eofId    = enterStandard(symTabStack, FUNCTION, "eof",    EOF);
        eolnId   = enterStandard(symTabStack, FUNCTION, "eoln",   EOLN);
        expId    = enterStandard(symTabStack, FUNCTION, "exp",    EXP);
        lnId     = enterStandard(symTabStack, FUNCTION, "ln",     LN);
        oddId    = enterStandard(symTabStack, FUNCTION, "odd",    ODD);
        ordId    = enterStandard(symTabStack, FUNCTION, "ord",    ODD);
        predId   = enterStandard(symTabStack, FUNCTION, "pred",   PRED);
        roundId  = enterStandard(symTabStack, FUNCTION, "round",  ROUND);
        sinId    = enterStandard(symTabStack, FUNCTION, "sin",    SIN);
        sqrId    = enterStandard(symTabStack, FUNCTION, "sqr",    SQR);
        sqrtId   = enterStandard(symTabStack, FUNCTION, "sqrt",   SQRT);
        succId   = enterStandard(symTabStack, FUNCTION, "succ",   SUCC);
        truncId  = enterStandard(symTabStack, FUNCTION, "trunc",  TRUNC);
    }

    /**
     * Enter a standard procedure or function into the symbol table stack.
     * @param symTabStack the symbol table stack to initialize.
     * @param defn either PROCEDURE or FUNCTION.
     * @param name the procedure or function name.
     */
    private static SymTabEntry enterStandard(SymTabStack symTabStack,
                                             Definition defn, String name,
                                             RoutineCode routineCode)
    {
        SymTabEntry procId = symTabStack.enterLocal(name);
        procId.setDefinition(defn);
        procId.setAttribute(ROUTINE_CODE, routineCode);

        return procId;
    }
```

Listing 11-11 shows the `parse()` method of the statement parser subclass `CallParser`.

Listing 11-11: Method `parse()` of class `CallParser`

```java
    /**
     * Parse a call to a declared procedure or function.
     * @param token the initial token.
     * @return the root node of the generated parse tree.
     * @throws Exception if an error occurred.
     */
    public ICodeNode parse(Token token)
        throws Exception
    {
        SymTabEntry pfId = symTabStack.lookup(token.getText().toLowerCase());
        RoutineCode routineCode = (RoutineCode) pfId.getAttribute(ROUTINE_CODE);
        StatementParser callParser = (routineCode == DECLARED) ||
                                     (routineCode == FORWARD)
                                         ? new CallDeclaredParser(this)
                                         : new CallStandardParser(this);

        return callParser.parse(token);
    }
```

Method `parse()` looks up the symbol table entry of the name of the routine being called. Depending on whether the routine is declared or forwarded or predefined, it creates a `CallDeclaredParser` or a `CallStandardParser` object and calls `callParser.parse()`.

Calls to Declared Procedures and Functions

Listing 11-12 shows the `parse()` method of class `CallDeclaredParser`.

Listing 11-12: Method `parse()` of class `CallDeclaredParser`

```java
    /**
     * Parse a call to a declared procedure or function.
     * @param token the initial token.
     * @return the root node of the generated parse tree.
     * @throws Exception if an error occurred.
     */
    public ICodeNode parse(Token token)
        throws Exception
    {
        // Create the CALL node.
        ICodeNode callNode = ICodeFactory.createICodeNode(CALL);
        SymTabEntry pfId = symTabStack.lookup(token.getText().toLowerCase());
        callNode.setAttribute(ID, pfId);
        callNode.setTypeSpec(pfId.getTypeSpec());

        token = nextToken();  // consume procedure or function identifier

        ICodeNode parmsNode = parseActualParameters(token, pfId,
```

```
                                              true, false, false);

        callNode.addChild(parmsNode);
        return callNode;
    }
```

Method `parse()` calls `parseActualParameters()`, which parses the actual parameters and returns the `PARAMETERS` node, or null if there are no parameters. The `CALL` node adopts the `PARAMETERS` node.

As an example, suppose you have the following Pascal declarations:

```
TYPE
    arr = ARRAY [1..5] OF real;

VAR
    i, m : integer;
    a : arr;
    v, y : real;
    t : boolean;

PROCEDURE proc(j, k : integer; VAR x, y, z : real; VAR v : arr;
               VAR p : boolean; ch : char);
    BEGIN
        ...
    END;
```

Then the procedure call

```
proc(i, -7 + m, a[m], v, y, a, t, &apos;r&apos;);
```

generates the parse tree shown in [Listing 11-13](#).

[Listing 11-13:](#) Parse tree generated by a call to the declared procedure `proc`

```
<CALL id="proc" level="1" line="81">
    <PARAMETERS>
        <VARIABLE id="i" level="1" type_id="integer" />
        <ADD type_id="integer">
            <NEGATE>
                <INTEGER_CONSTANT value="7" type_id="integer" />
            </NEGATE>
            <VARIABLE id="m" level="1" type_id="integer" />
        </ADD>
        <VARIABLE id="a" level="1" type_id="real">
            <SUBSCRIPTS type_id="real">
                <VARIABLE id="m" level="1" type_id="integer" />
            </SUBSCRIPTS>
        </VARIABLE>
        <VARIABLE id="v" level="1" type_id="real" />
        <VARIABLE id="y" level="1" type_id="real" />
        <VARIABLE id="a" level="1" type_id="arr" />
        <VARIABLE id="t" level="1" type_id="boolean" />
        <STRING_CONSTANT value="r" type_id="char" />
    </PARAMETERS>
</CALL>
```

Calls to the Standard Procedures and Functions

[Listing 11-14](#) shows the `parse()` method of class `CallStandardParser`. It calls the appropriate private parsing method to parse each call to a standard procedure or function.

[Listing 11-14:](#) Method `parse()` of class `CallStandardParser`

```
/**
 * Parse a call to a declared procedure or function.
 * @param token the initial token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token)
    throws Exception
{
    ICodeNode callNode = ICodeFactory.createICodeNode(CALL);
    SymTabEntry pfId = symTabStack.lookup(token.getText().toLowerCase());
    RoutineCode routineCode = (RoutineCode) pfId.getAttribute(ROUTINE_CODE);
    callNode.setAttribute(ID, pfId);

    token = nextToken();  // consume procedure or function identifier

    switch ((RoutineCodeImpl) routineCode) {
        case READ:
        case READLN:  return parseReadReadln(token, callNode, pfId);

        case WRITE:
        case WRITELN: return parseWriteWriteln(token, callNode, pfId);

        case EOF:
        case EOLN:    return parseEofEoln(token, callNode, pfId);

        case ABS:
        case SQR:     return parseAbsSqr(token, callNode, pfId);

        case ARCTAN:
        case COS:
        case EXP:
        case LN:
        case SIN:
        case SQRT:    return parseArctanCosExpLnSinSqrt(token, callNode,
                                                        pfId);

        case PRED:
        case SUCC:    return parsePredSucc(token, callNode, pfId);

        case CHR:     return parseChr(token, callNode, pfId);
        case ODD:     return parseOdd(token, callNode, pfId);
        case ORD:     return parseOrd(token, callNode, pfId);

        case ROUND:
        case TRUNC:   return parseRoundTrunc(token, callNode, pfId);
```

```
            default:     return null;  // should never get here
        }
    }
}
```

 [Listing 11-15](#) shows these *ad hoc* private parsing methods.

<u>Listing 11-15:</u> Private parsing methods of class `CallStandardParser`

```java
    /**
     * Parse a call to read or readln.
     * @param token the current token.
     * @param callNode the CALL node.
     * @param pfId the symbol table entry of the standard routine name.
     * @return ICodeNode the CALL node.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseReadReadln(Token token, ICodeNode callNode,
                                      SymTabEntry pfId)
        throws Exception
    {
        // Parse any actual parameters.
        ICodeNode parmsNode = parseActualParameters(token, pfId,
                                                    false, true, false);
        callNode.addChild(parmsNode);

        // Read must have parameters.
        if ((pfId == Predefined.readId) &&
            (callNode.getChildren().size() == 0))
        {
            errorHandler.flag(token, WRONG_NUMBER_OF_PARMS, this);
        }

        return callNode;
    }

    /**
     * Parse a call to write or writeln.
     * @param token the current token.
     * @param callNode the CALL node.
     * @param pfId the symbol table entry of the standard routine name.
     * @return ICodeNode the CALL node.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseWriteWriteln(Token token, ICodeNode callNode,
                                        SymTabEntry pfId)
        throws Exception
    {
        // Parse any actual parameters.
        ICodeNode parmsNode = parseActualParameters(token, pfId,
                                                    false, false, true);
        callNode.addChild(parmsNode);

        // Write must have parameters.
        if ((pfId == Predefined.writeId) &&
            (callNode.getChildren().size() == 0))
        {
            errorHandler.flag(token, WRONG_NUMBER_OF_PARMS, this);
        }

        return callNode;
    }

    /**
     * Parse a call to eof or eoln.
     * @param token the current token.
     * @param callNode the CALL node.
     * @param pfId the symbol table entry of the standard routine name.
     * @return ICodeNode the CALL node.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseEofEoln(Token token, ICodeNode callNode,
                                   SymTabEntry pfId)
        throws Exception
    {
        // Parse any actual parameters.
        ICodeNode parmsNode = parseActualParameters(token, pfId,
                                                    false, false, false);
        callNode.addChild(parmsNode);

        // There should be no actual parameters.
        if (checkParmCount(token, parmsNode, 0)) {
            callNode.setTypeSpec(Predefined.booleanType);
        }

        return callNode;
    }

    /**
     * Parse a call to abs or sqr.
     * @param token the current token.
     * @param callNode the CALL node.
     * @param pfId the symbol table entry of the standard routine name.
     * @return ICodeNode the CALL node.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseAbsSqr(Token token, ICodeNode callNode,
                                  SymTabEntry pfId)
        throws Exception
    {
        // Parse any actual parameters.
        ICodeNode parmsNode = parseActualParameters(token, pfId,
                                                    false, false, false);
        callNode.addChild(parmsNode);

        // There should be one integer or real parameter.
        // The function return type is the parameter type.
        if (checkParmCount(token, parmsNode, 1)) {
            TypeSpec argType =
                parmsNode.getChildren().get(0).getTypeSpec().baseType();

            if ((argType == Predefined.integerType) ||
                (argType == Predefined.realType)) {
                callNode.setTypeSpec(argType);
            }
            else {
                errorHandler.flag(token, INVALID_TYPE, this);
            }
        }

        return callNode;
```

```
        }

    /**
     * Parse a call to arctan, cos, exp, ln, sin, or sqrt.
     * @param token the current token.
     * @param callNode the CALL node.
     * @param pfId the symbol table entry of the standard routine name.
     * @return ICodeNode the CALL node.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseArctanCosExpLnSinSqrt(Token token,
                                                 ICodeNode callNode,
                                                 SymTabEntry pfId)
        throws Exception
    {
        // Parse any actual parameters.
        ICodeNode parmsNode = parseActualParameters(token, pfId,
                                                    false, false, false);
        callNode.addChild(parmsNode);

        // There should be one integer or real parameter.
        // The function return type is real.
        if (checkParmCount(token, parmsNode, 1)) {
            TypeSpec argType =
                parmsNode.getChildren().get(0).getTypeSpec().baseType();

            if ((argType == Predefined.integerType) ||
                (argType == Predefined.realType)) {
                callNode.setTypeSpec(Predefined.realType);
            }
            else {
                errorHandler.flag(token, INVALID_TYPE, this);
            }
        }

        return callNode;
    }

    /**
     * Parse a call to pred or succ.
     * @param token the current token.
     * @param callNode the CALL node.
     * @param pfId the symbol table entry of the standard routine name.
     * @return ICodeNode the CALL node.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parsePredSucc(Token token, ICodeNode callNode,
                                    SymTabEntry pfId)
        throws Exception
    {
        // Parse any actual parameters.
        ICodeNode parmsNode = parseActualParameters(token, pfId,
                                                    false, false, false);
        callNode.addChild(parmsNode);

        // There should be one integer or enumeration parameter.
        // The function return type is the parameter type.
        if (checkParmCount(token, parmsNode, 1)) {
            TypeSpec argType =
                parmsNode.getChildren().get(0).getTypeSpec().baseType();

            if ((argType == Predefined.integerType) ||
                (argType.getForm() == ENUMERATION))
            {
                callNode.setTypeSpec(argType);
            }
            else {
                errorHandler.flag(token, INVALID_TYPE, this);
            }
        }

        return callNode;
    }

    /**
     * Parse a call to chr.
     * @param token the current token.
     * @param callNode the CALL node.
     * @param pfId the symbol table entry of the standard routine name.
     * @return ICodeNode the CALL node.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseChr(Token token, ICodeNode callNode,
                               SymTabEntry pfId)
        throws Exception
    {
        // Parse any actual parameters.
        ICodeNode parmsNode = parseActualParameters(token, pfId,
                                                    false, false, false);
        callNode.addChild(parmsNode);

        // There should be one integer parameter.
        // The function return type is character.
        if (checkParmCount(token, parmsNode, 1)) {
            TypeSpec argType =
                parmsNode.getChildren().get(0).getTypeSpec().baseType();

            if (argType == Predefined.integerType) {
                callNode.setTypeSpec(Predefined.charType);
            }
            else {
                errorHandler.flag(token, INVALID_TYPE, this);
            }
        }

        return callNode;
    }

    /**
     * Parse a call to odd.
     * @param token the current token.
     * @param callNode the CALL node.
     * @param pfId the symbol table entry of the standard routine name.
     * @return ICodeNode the CALL node.
     * @throws Exception if an error occurred.
     */
    private ICodeNode parseOdd(Token token, ICodeNode callNode,
                               SymTabEntry pfId)
        throws Exception
    {
        // Parse any actual parameters.
        ICodeNode parmsNode = parseActualParameters(token, pfId,
```

```
                                                          false, false, false);
            callNode.addChild(parmsNode);

            // There should be one integer parameter.
            // The function return type is boolean.
            if (checkParmCount(token, parmsNode, 1)) {
                TypeSpec argType =
                    parmsNode.getChildren().get(0).getTypeSpec().baseType();

                if (argType == Predefined.integerType) {
                    callNode.setTypeSpec(Predefined.booleanType);
                }
                else {
                    errorHandler.flag(token, INVALID_TYPE, this);
                }
            }

            return callNode;
        }

        /**
         * Parse a call to ord.
         * @param token the current token.
         * @param callNode the CALL node.
         * @param pfId the symbol table entry of the standard routine name.
         * @return ICodeNode the CALL node.
         * @throws Exception if an error occurred.
         */
        private ICodeNode parseOrd(Token token, ICodeNode callNode,
                                   SymTabEntry pfId)
            throws Exception
        {
            // Parse any actual parameters.
            ICodeNode parmsNode = parseActualParameters(token, pfId,
                                                         false, false, false);
            callNode.addChild(parmsNode);

            // There should be one character or enumeration parameter.
            // The function return type is integer.
            if (checkParmCount(token, parmsNode, 1)) {
                TypeSpec argType =
                    parmsNode.getChildren().get(0).getTypeSpec().baseType();

                if ((argType == Predefined.charType) ||
                    (argType.getForm() == ENUMERATION)) {
                    callNode.setTypeSpec(Predefined.integerType);
                }
                else {
                    errorHandler.flag(token, INVALID_TYPE, this);
                }
            }

            return callNode;
        }

        /**
         * Parse a call to round or trunc.
         * @param token the current token.
         * @param callNode the CALL node.
         * @param pfId the symbol table entry of the standard routine name.
         * @return ICodeNode the CALL node.
         * @throws Exception if an error occurred.
         */
        private ICodeNode parseRoundTrunc(Token token, ICodeNode callNode,
                                          SymTabEntry pfId)
            throws Exception
        {
            // Parse any actual parameters.
            ICodeNode parmsNode = parseActualParameters(token, pfId,
                                                         false, false, false);
            callNode.addChild(parmsNode);

            // There should be one real parameter.
            // The function return type is integer.
            if (checkParmCount(token, parmsNode, 1)) {
                TypeSpec argType =
                    parmsNode.getChildren().get(0).getTypeSpec().baseType();

                if (argType == Predefined.realType) {
                    callNode.setTypeSpec(Predefined.integerType);
                }
                else {
                    errorHandler.flag(token, INVALID_TYPE, this);
                }
            }

            return callNode;
        }

        /**
         * Check the number of actual parameters.
         * @param token the current token.
         * @param parmsNode the PARAMETERS node.
         * @param count the correct number of parameters.
         * @return true if the count is correct.
         */
        private boolean checkParmCount(Token token, ICodeNode parmsNode, int count)
        {
            if ( ((parmsNode == null) && (count == 0)) ||
                 (parmsNode.getChildren().size() == count) ) {
                return true;
            }
            else {
                errorHandler.flag(token, WRONG_NUMBER_OF_PARMS, this);
                return false;
            }
        }
```

Calls to the standard procedures read, readln, write, and writeln can variable numbers of actual parameters; each call to read and write must have at least one parameter. Calls to the standard functions eof and eoln has no actual parameters, and calls to the other standard functions must each have exactly one actual parameter.

In each parsing method, a call to parseActualParameters() parses the actual parameters and returns a PARAMETERS node, or null if there are no parameters. The CALL node adopts the PARAMETERS node. Method checkParmCount() verifies that the number of actual parameters in a call is correct.

Functions eof and eoln each returns a boolean value. Functions abs and sqr each has an integer or a real parameter, and the return value type is the same as the parameter type. Functions arctan, cos, exp, ln, sin, and sqrt each has an integer or a real parameter, and

the return value is always real. Functions `pred` and `succ` each has an integer or an enumeration parameter, and the return value type is the same as the parameter type. Function `chr` has an integer parameter and returns a character value. Function `odd` has an integer parameter and returns a boolean value. Function `ord` has a character or an enumeration value and returns an integer value. Functions `round` and `trunc` each has a real parameter and returns an integer value.

Actual Parameter Lists

If there are actual parameters in a call to either a declared routine or to a standard routine, method `parseActualParameters()` in class `CallParser` parses the parameters. The boolean arguments `isDeclared`, `isReadReadln`, `isWriteWriteln` indicate whether the call is to a declared routine, to either standard procedure `read` or `readln`, or to either standard procedure `write` or `writeln`, respectively. Only one will be true at a time. See .

Listing 11-16: Method `parseActualParameters()` of class `CallParser`

```java
/**
 * Parse the actual parameters of a procedure or function call.
 * @param token the current token.
 * @param pfId the symbol table entry of the procedure or function name.
 * @param isDeclared true if parsing actual parms of a declared routine.
 * @param isReadReadln true if parsing actual parms of read or readln.
 * @param isWriteWriteln true if parsing actual parms of write or writeln.
 * @return the PARAMETERS node, or null if there are no actual parameters.
 * @throws Exception if an error occurred.
 */
protected ICodeNode parseActualParameters(Token token, SymTabEntry pfId,
                                          boolean isDeclared,
                                          boolean isReadReadln,
                                          boolean isWriteWriteln)
    throws Exception
{
    ExpressionParser expressionParser = new ExpressionParser(this);
    ICodeNode parmsNode = ICodeFactory.createICodeNode(PARAMETERS);
    ArrayList<SymTabEntry> formalParms = null;
    int parmCount = 0;
    int parmIndex = -1;

    if (isDeclared) {
        formalParms =
            (ArrayList<SymTabEntry>) pfId.getAttribute(ROUTINE_PARMS);
        parmCount = formalParms != null ? formalParms.size() : 0;
    }

    if (token.getType() != LEFT_PAREN) {
        if (parmCount != 0) {
            errorHandler.flag(token, WRONG_NUMBER_OF_PARMS, this);
        }

        return null;
    }

    token = nextToken();  // consume opening (

    // Loop to parse each actual parameter.
    while (token.getType() != RIGHT_PAREN) {
        ICodeNode actualNode = expressionParser.parse(token);

        // Declared procedure or function: Check the number of actual
        // parameters, and check each actual parameter against the
        // corresponding formal parameter.
        if (isDeclared) {
            if (++parmIndex < parmCount) {
                SymTabEntry formalId = formalParms.get(parmIndex);
                checkActualParameter(token, formalId, actualNode);
            }
            else if (parmIndex == parmCount) {
                errorHandler.flag(token, WRONG_NUMBER_OF_PARMS, this);
            }
        }

        // read or readln: Each actual parameter must be a variable that is
        //                 a scalar, boolean, or subrange of integer.
        else if (isReadReadln) {
            TypeSpec type = actualNode.getTypeSpec();
            TypeForm form = type.getForm();

            if (! (   (actualNode.getType() == ICodeNodeTypeImpl.VARIABLE)
                   && ( (form == SCALAR) ||
                        (type == Predefined.booleanType) ||
                        ( (form == SUBRANGE) &&
                          (type.baseType() == Predefined.integerType) ) )
                  )
               )
            {
                errorHandler.flag(token, INVALID_VAR_PARM, this);
            }
        }

        // write or writeln: The type of each actual parameter must be a
        // scalar, boolean, or a Pascal string. Parse any field width and
        // precision.
        else if (isWriteWriteln) {

            // Create a WRITE_PARM node which adopts the expression node.
            ICodeNode exprNode = actualNode;
            actualNode = ICodeFactory.createICodeNode(WRITE_PARM);
            actualNode.addChild(exprNode);

            TypeSpec type = exprNode.getTypeSpec().baseType();
            TypeForm form = type.getForm();

            if (! ( (form == SCALAR) || (type == Predefined.booleanType) ||
                    (type.isPascalString())
                  )
               )
            {
                errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
            }

            // Optional field width.
            token = currentToken();
            actualNode.addChild(parseWriteSpec(token));
```

```
                // Optional precision.
                token = currentToken();
                actualNode.addChild(parseWriteSpec(token));
            }

            parmsNode.addChild(actualNode);
            token = synchronize(COMMA_SET);
            TokenType tokenType = token.getType();

            // Look for the comma.
            if (tokenType == COMMA) {
                token = nextToken();  // consume ,
            }
            else if (ExpressionParser.EXPR_START_SET.contains(tokenType)) {
                errorHandler.flag(token, MISSING_COMMA, this);
            }
            else if (tokenType != RIGHT_PAREN) {
                token = synchronize(ExpressionParser.EXPR_START_SET);
            }
        }

        token = nextToken();  // consume closing )

        if ((parmsNode.getChildren().size() == 0) ||
            (isDeclared && (parmIndex != parmCount-1)))
        {
            errorHandler.flag(token, WRONG_NUMBER_OF_PARMS, this);
        }

        return parmsNode;
    }
```

Method `parseActualParameters()` creates and returns a `PARAMETERS` node if there are actual parameters, or it returns null if there are no parameters. It loops to parse each actual parameter, and the `PARAMETERS` node adopts each parameter's expression parse tree.

If the call is to a declared routine, the method calls `checkActualParameter()` to check each actual parameter against the corresponding formal parameter, and it verifies that the number of actual parameters matches the number of formal parameters. For calls to the standard procedures `read` and `readln`, the method checks that each actual parameter is a variable whose type is scalar, boolean, or subrange of integer.

Calls to the standard procedures `write` and `writeln` are unique. Each actual parameter must be an expression whose type scalar, boolean, or a Pascal string. Each parameter can be followed by a colon and an integer constant whose value is the width of the output field in characters. The field width can be followed by a second colon and an integer constant whose value is the precision (number of fraction digits) when outputting a real value. For example:

```
writeln('The square root of ', number:5, ' is ', sqrt(number):10:6);
```

Therefore, for each parameter, method `parseActualParameters()` creates a `WRITE_PARM` node which adopts the actual parameter's expression parse tree as its first child. If there is a field width, the `WRITE_PARM` node adopts it as its second child, and if there is a precision, the node adopts it as its third child. Listing 11-17 shows the parse tree generated by the above call to `writeln`.

Listing 11-17: Parse tree generated by a call to the standard procedure `writeln`

```
<CALL line="11" id="writeln" level="0">
    <PARAMETERS>
        <WRITE_PARM>
            <STRING_CONSTANT value="The square root of "
                type_id="$anon_4cb088b" />
        </WRITE_PARM>
        <WRITE_PARM>
            <VARIABLE id="number" level="1" type_id="integer" />
            <INTEGER_CONSTANT value="5" type_id="integer" />
        </WRITE_PARM>
        <WRITE_PARM>
            <STRING_CONSTANT value=" is " type_id="$anon_4cb0887" />
        </WRITE_PARM>
        <WRITE_PARM>
            <CALL id="sqrt" level="0" type_id="real">
                <PARAMETERS>
                    <VARIABLE id="number" level="1" type_id="integer" />
                </PARAMETERS>
            </CALL>
            <INTEGER_CONSTANT value="10" type_id="integer" />
            <INTEGER_CONSTANT value="6" type_id="integer" />
        </WRITE_PARM>
    </PARAMETERS>
</CALL>
```

Listing 11-18 shows method `checkActualParameter()`.

Listing 11-18: Method `checkActualParameter()` of class `CallParser`

```
/**
 * Check an actual parameter against the corresponding formal parameter.
 * @param token the current token.
 * @param formalId the symbol table entry of the formal parameter.
 * @param actualNode the parse tree node of the actual parameter.
 */
private void checkActualParameter(Token token, SymTabEntry formalId,
                                  ICodeNode actualNode)
{
    Definition formalDefn = formalId.getDefinition();
    TypeSpec formalType = formalId.getTypeSpec();
    TypeSpec actualType = actualNode.getTypeSpec();

    // VAR parameter: The actual parameter must be a variable of the same
    //                type as the formal parameter.
    if (formalDefn == VAR_PARM) {
        if ((actualNode.getType() != ICodeNodeTypeImpl.VARIABLE) ||
            (actualType != formalType))
        {
            errorHandler.flag(token, INVALID_VAR_PARM, this);
        }
    }

    // Value parameter: The actual parameter must be assignment-compatible
    //                  with the formal parameter.
    else if (!TypeChecker.areAssignmentCompatible(formalType, actualType)) {
        errorHandler.flag(token, INCOMPATIBLE_TYPES, this);
    }
}
```

Method `checkActualParameter()` checks each actual parameter against the corresponding formal parameter. If the parameter is a `VAR` parameter (passed by reference), the actual parameter must be a variable of the same type as the formal parameter. If the parameter is passed by value, then the actual parameter type must be assignment compatible with the formal parameter type.

Listing 11-19 shows method `parseWriteSpec()`, which parses the optional field width and precision values after each actual parameter of a call to the standard procedures `write` and `writeln`.

Listing 11-19: Method `parseWriteSpec()` of class `CallParser`

```
/**
 * Parse the field width or the precision for an actual parameter
 * of a call to write or writeln.
 * @param token the current token.
 * @return the INTEGER_CONSTANT node or null
 * @throws Exception if an error occurred.
 */
private ICodeNode parseWriteSpec(Token token)
    throws Exception
{
    if (token.getType() == COLON) {
        token = nextToken();  // consume :

        ExpressionParser expressionParser = new ExpressionParser(this);
        ICodeNode specNode = expressionParser.parse(token);

        if (specNode.getType() == INTEGER_CONSTANT) {
            return specNode;
        }
        else {
            errorHandler.flag(token, INVALID_NUMBER, this);
            return null;
        }
    }
    else {
        return null;
    }
}
```

Method `parseWriteSpec()` looks for a `:` token and then parses an integer constant.

Modify class `StatementParser` to parse procedure calls and assignments to formal parameters and function identifiers. Listing 11-20 shows the new version of `case IDENTIFIER` in the `parse()` method's `switch` statement.

Listing 11-20: Modifications to method `parse()` of class `StatementParser`

```
case IDENTIFIER: {
    String name = token.getText().toLowerCase();
    SymTabEntry id = symTabStack.lookup(name);
    Definition idDefn = id != null ? id.getDefinition()
                                    : UNDEFINED;

    // Assignment statement or procedure call.
    switch ((DefinitionImpl) idDefn) {

        case VARIABLE:
        case VALUE_PARM:
        case VAR_PARM:
        case UNDEFINED: {
            AssignmentStatementParser assignmentParser =
                new AssignmentStatementParser(this);
            statementNode = assignmentParser.parse(token);
            break;
        }

        case FUNCTION: {
            AssignmentStatementParser assignmentParser =
                new AssignmentStatementParser(this);
            statementNode =
                assignmentParser.parseFunctionNameAssignment(token);
            break;
        }

        case PROCEDURE: {
            CallParser callParser = new CallParser(this);
            statementNode = callParser.parse(token);
            break;
        }

        default: {
            errorHandler.flag(token, UNEXPECTED_TOKEN, this);
            token = nextToken();  // consume identifier
        }
    }

    break;
}
```

When the current token is a function name, method `parse()` calls `assignmentParser.parseFunctionNameAssignment()`. If the current token is a procedure name, the method instead calls `callParser.parse()`.

Listing 11-21 shows the additions to class `AssignmentStatementParser`.

Listing 11-21: Method `parseFunctionNameAssignment()` of class `AssignmentStatementParser`

```
// Set to true to parse a function name
// as the target of an assignment.
private boolean isFunctionTarget = false;

/**
 * Parse an assignment to a function name.
 * @param token Token
 * @return ICodeNode
 * @throws Exception
 */
public ICodeNode parseFunctionNameAssignment(Token token)
    throws Exception
{
    isFunctionTarget = true;
    return parse(token);
}
```

Method `parseFunctionNameAssignment()` sets the private field `isFunctionTarget` to true before calling `parse()`. Method `parse()` requires a small change:

```
// Parse the target variable.
    VariableParser variableParser = new VariableParser(this);
    ICodeNode targetNode = isFunctionTarget
                         ? variableParser.parseFunctionNameTarget(token)
                         : variableParser.parse(token);
    TypeSpec targetType = targetNode != null ? targetNode.getTypeSpec()
                                             : Predefined.undefinedType;
```

Modify class `VariableParser`. See [Listing 11-22](#).

Listing 11-22: Methods `parseFunctionNameTarget()` and `parse()` of class `VariableParser`

```
// Set to true to parse a function name
// as the target of an assignment.
private boolean isFunctionTarget = false;

/**
 * Parse a function name as the target of an assignment statement.
 * @param token the initial token.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parseFunctionNameTarget(Token token)
    throws Exception
{
    isFunctionTarget = true;
    return parse(token);
}

/**
 * Parse a variable.
 * @param token the initial token.
 * @param variableId the symbol table entry of the variable identifier.
 * @return the root node of the generated parse tree.
 * @throws Exception if an error occurred.
 */
public ICodeNode parse(Token token, SymTabEntry variableId)
    throws Exception
{
    // Check how the variable is defined.
    Definition defnCode = variableId.getDefinition();
    if (! ( (defnCode == VARIABLE) || (defnCode == VALUE_PARM) ||
            (defnCode == VAR_PARM) ||
            (isFunctionTarget && (defnCode == FUNCTION) )
          )
       )
    {
        errorHandler.flag(token, INVALID_IDENTIFIER_USAGE, this);
    }

    variableId.appendLineNumber(token.getLineNumber());

    ICodeNode variableNode =
        ICodeFactory.createICodeNode(ICodeNodeTypeImpl.VARIABLE);
    variableNode.setAttribute(ID, variableId);

    token = nextToken();  // consume the identifier
    TypeSpec variableType = variableId.getTypeSpec();

    if (!isFunctionTarget) {

        // Parse array subscripts or record fields.
        while (SUBSCRIPT_FIELD_START_SET.contains(token.getType())) {
            ICodeNode subFldNode = token.getType() == LEFT_BRACKET
                                 ? parseSubscripts(variableType)
                                 : parseField(variableType);
            token = currentToken();

            // Update the variable&apos;s type.
            // The variable node adopts the SUBSCRIPTS or FIELD node.
            variableType = subFldNode.getTypeSpec();
            variableNode.addChild(subFldNode);
        }
    }

    variableNode.setTypeSpec(variableType);
    return variableNode;
}
```

Method `parseFunctionNameTarget()` sets the private field `isFunctionTarget` to true before calling `parse()`. Method `parse()` can now also parse a formal parameter name and the name of a declared function. Because a Pascal function cannot return an array or a record, a function name cannot be followed by array subscripts or record fields.

Modify class `ExpressionParser` to parse function calls. In the `parseIdentifier()` method's `switch` statement, add:

```
case FUNCTION: {
    CallParser callParser = new CallParser(this);
    rootNode = callParser.parse(token);
    break;
}
```

Program 11: Pascal Syntax Checker IV

With the addition of the classes in the front end to parse procedure and function declarations and calls, you can now check the syntax of entire programs written in the Pascal subset you've been using.

[Listing 11-23](#) shows the results of "compiling" a source program that declares and makes calls to procedures and functions nested at various levels. The command line is similar to:

```
java -classpath classes Pascal compile -ix routines.pas
```

As before, the command is to compile since you haven't written all the executors in the back end.

Listing 11-23: Output from compiling a Pascal program

```
001 PROGRAM RoutinesTest (input, output);
002
003 CONST
004     five = 5;
005
006 TYPE
007     enum = (alpha, beta, gamma);
008     arr = ARRAY [1..five] OF real;
009
010 VAR
011     e, k : enum;
012     i, m : integer;
013     a : arr;
014     v, y : real;
015     t : boolean;
016
017 PROCEDURE proc(j, k : integer; VAR x, y, z : real; VAR v : arr;
018               VAR p : boolean; ch : char);
019
020     VAR
021         i : real;
022
023     BEGIN
024         i := 7*k;
025         x := 3.14;
026     END;
027
028 FUNCTION forwarded(m : integer; VAR t : real) : real; forward;
029
030 FUNCTION func(VAR x : real; i : real; n : integer) : real;
031
032     VAR
033         z : real;
034         p, q : boolean;
035
036     PROCEDURE nested(VAR n, m : integer);

037
038         CONST
039             ten = 10;
040
041         TYPE
042             subrange = five..ten;
043
044         VAR
045             a, b, c : integer;
046             s : subrange;
047
048         PROCEDURE deeply;
049
050             VAR
051                 w : real;
052
053             BEGIN
054                 w := i;
055                 nested(a, m);
056                 w := forwarded(b, w);
057             END;
058
059         BEGIN {nested}
060             s := m;
061             deeply;
062             a := s
063         END {nested};
064
065     BEGIN {func}
066         p := true;
067         q := false;
068         x := i*z - func(v, -3.15159, five) + n/m;
069         func := x;
070     END {func};
071
072 FUNCTION forwarded;
073
074     VAR
075         n : integer;
076
077     BEGIN
078         forwarded := n*y - t;
079     END;
080
081 BEGIN {RoutinesTest}
082     e := beta;
083     proc(i, -7 + m, a[m], v, y, a, t, &apos;r&apos;);
084 END {RoutinesTest}.

                84 source lines.
                 0 syntax errors.
                0.13 seconds total parsing time.


===== CROSS-REFERENCE TABLE =====

*** PROGRAM routinestest ***


Identifier     Line numbers   Type specification
----------     ------------   ------------------
a              013 083 083
                              Defined as: variable
                              Scope nesting level: 1
                              Type form = array, Type id = arr
alpha          007
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum
                              Value = 0
arr            008 013 017
                              Defined as: type
                              Scope nesting level: 1
                              Type form = array, Type id = arr
                              --- INDEX TYPE ---
                              Type form = subrange, Type id = <unnamed>
                              --- Base type ---
                              Type form = scalar, Type id = integer
                              Range = 1..5
                              --- ELEMENT TYPE ---
                              Type form = scalar, Type id = real
                              5 elements
beta           007 082
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum
                              Value = 1
e              011 082
```

```
                              Defined as: variable
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum
enum            007 011
                              Defined as: type
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum
                              --- Enumeration constants ---
                                        alpha = 0
                                         beta = 1
                                        gamma = 2
five            004 008 042 068
                              Defined as: constant
                              Scope nesting level: 1
                              Type form = scalar, Type id = integer
                              Value = 5
forwarded       078
                              Defined as: function
                              Scope nesting level: 1
                              Type form = scalar, Type id = real
func            069
                              Defined as: function
                              Scope nesting level: 1
                              Type form = scalar, Type id = real
gamma           007
                              Defined as: enumeration constant
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum
                              Value = 2
i               012 083
                              Defined as: variable
                              Scope nesting level: 1
                              Type form = scalar, Type id = integer
input           001
                              Defined as: program parameter
                              Scope nesting level: 1
k               011
                              Defined as: variable
                              Scope nesting level: 1
                              Type form = enumeration, Type id = enum
m               012 068 083 083
                              Defined as: variable
                              Scope nesting level: 1
                              Type form = scalar, Type id = integer
output          001
                              Defined as: program parameter
                              Scope nesting level: 1
proc
                              Defined as: procedure
                              Scope nesting level: 1
t               015 083
                              Defined as: variable
                              Scope nesting level: 1
                              Type form = enumeration, Type id = boolean
v               014 068 083
                              Defined as: variable
                              Scope nesting level: 1
                              Type form = scalar, Type id = real
y               014 078 083
                              Defined as: variable
                              Scope nesting level: 1
                              Type form = scalar, Type id = real


*** PROCEDURE proc ***

Identifier      Line numbers   Type specification
----------      ------------   ------------------
ch              018
                              Defined as: value parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = char
i               021 024
                              Defined as: variable
                              Scope nesting level: 2
                              Type form = scalar, Type id = real
j               017
                              Defined as: value parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = integer
k               017 024
                              Defined as: value parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = integer
p               018
                              Defined as: VAR parameter
                              Scope nesting level: 2
                              Type form = enumeration, Type id = boolean
v               017
                              Defined as: VAR parameter
                              Scope nesting level: 2
                              Type form = array, Type id = arr
x               017 025
                              Defined as: VAR parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = real
y               017
                              Defined as: VAR parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = real
z               017
                              Defined as: VAR parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = real


*** FUNCTION forwarded ***

Identifier      Line numbers   Type specification
----------      ------------   ------------------
m               028
                              Defined as: value parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = integer
n               075 078
                              Defined as: variable
                              Scope nesting level: 2
                              Type form = scalar, Type id = integer
t               028 078
                              Defined as: VAR parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = real


*** FUNCTION func ***
```

```
Identifier     Line numbers   Type specification
----------     ------------   ------------------
i              030 054 068
                              Defined as: value parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = real
n              030 068
                              Defined as: value parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = integer
nested
                              Defined as: procedure
                              Scope nesting level: 2
p              034 066
                              Defined as: variable
                              Scope nesting level: 2
                              Type form = enumeration, Type id = boolean
q              034 067
                              Defined as: variable
                              Scope nesting level: 2
                              Type form = enumeration, Type id = boolean
x              030 068 069
                              Defined as: VAR parameter
                              Scope nesting level: 2
                              Type form = scalar, Type id = real
z              033 068
                              Defined as: variable
                              Scope nesting level: 2
                              Type form = scalar, Type id = real


*** PROCEDURE nested ***

Identifier     Line numbers   Type specification
----------     ------------   ------------------
a              045 055 062
                              Defined as: variable
                              Scope nesting level: 3
                              Type form = scalar, Type id = integer
b              045 056
                              Defined as: variable
                              Scope nesting level: 3
                              Type form = scalar, Type id = integer
c              045
                              Defined as: variable
                              Scope nesting level: 3
                              Type form = scalar, Type id = integer
deeply
                              Defined as: procedure
                              Scope nesting level: 3
m              036 055 060
                              Defined as: VAR parameter
                              Scope nesting level: 3
                              Type form = scalar, Type id = integer
n              036
                              Defined as: VAR parameter
                              Scope nesting level: 3
                              Type form = scalar, Type id = integer
s              046 060 062
                              Defined as: variable
                              Scope nesting level: 3
                              Type form = subrange, Type id = subrange
subrange       042 046
                              Defined as: type
                              Scope nesting level: 3
                              Type form = subrange, Type id = subrange
                              --- Base type ---
                              Type form = scalar, Type id = integer
                              Range = 5..10
ten            039 042
                              Defined as: constant
                              Scope nesting level: 3
                              Type form = scalar, Type id = integer
                              Value = 10


*** PROCEDURE deeply ***

Identifier     Line numbers   Type specification
----------     ------------   ------------------
w              051 054 056 056
                              Defined as: variable
                              Scope nesting level: 4
                              Type form = scalar, Type id = real


===== INTERMEDIATE CODE =====

*** PROGRAM routinestest ***

<COMPOUND line="81">
    <ASSIGN line="82" type_id="enum">
        <VARIABLE id="e" level="1" type_id="enum" />
        <INTEGER_CONSTANT value="1" type_id="enum" />
    </ASSIGN>
    <CALL line="83" id="proc" level="1">
        <PARAMETERS>
            <VARIABLE id="i" level="1" type_id="integer" />
            <ADD type_id="integer">
                <NEGATE>
                    <INTEGER_CONSTANT value="7" type_id="integer" />
                </NEGATE>
                <VARIABLE id="m" level="1" type_id="integer" />
            </ADD>
            <VARIABLE id="a" level="1" type_id="real">
                <SUBSCRIPTS type_id="real">
                    <VARIABLE id="m" level="1" type_id="integer" />
                </SUBSCRIPTS>
            </VARIABLE>
            <VARIABLE id="v" level="1" type_id="real" />
            <VARIABLE id="y" level="1" type_id="real" />
            <VARIABLE id="a" level="1" type_id="arr" />
            <VARIABLE id="t" level="1" type_id="boolean" />
            <STRING_CONSTANT value="r" type_id="char" />
        </PARAMETERS>
    </CALL>
</COMPOUND>


*** PROCEDURE proc ***

<COMPOUND line="23">
    <ASSIGN line="24" type_id="real">
        <VARIABLE id="i" level="2" type_id="real" />
        <MULTIPLY type_id="integer">
```
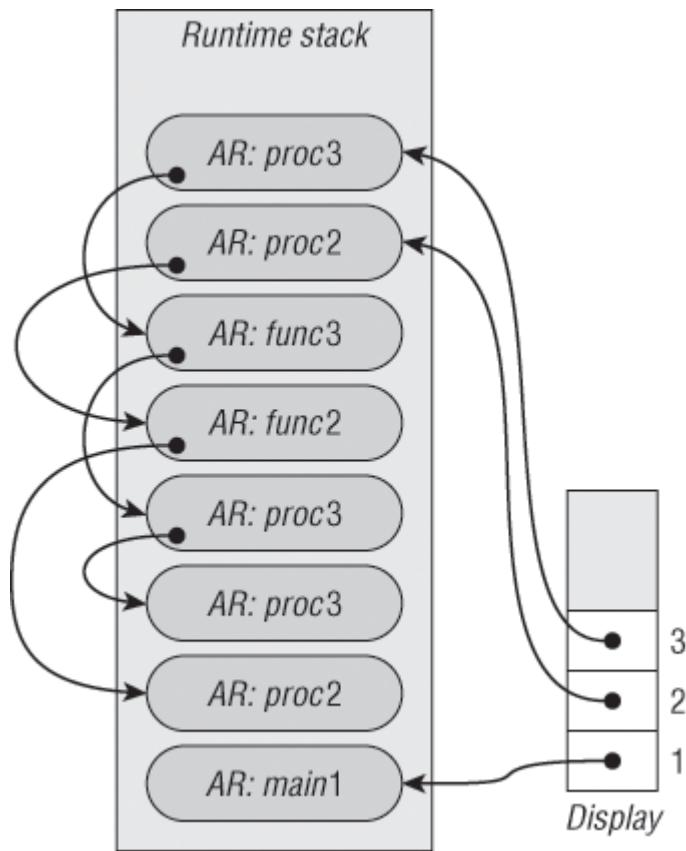
```
                <INTEGER_CONSTANT value="7" type_id="integer" />
                <VARIABLE id="k" level="2" type_id="integer" />
            </MULTIPLY>
        </ASSIGN>
        <ASSIGN line="25" type_id="real">
            <VARIABLE id="x" level="2" type_id="real" />
            <REAL_CONSTANT value="3.14" type_id="real" />
        </ASSIGN>
</COMPOUND>


*** FUNCTION forwarded ***


<COMPOUND line="77">
    <ASSIGN line="78" type_id="real">
        <VARIABLE id="forwarded" level="1" type_id="real" />
        <SUBTRACT type_id="real">
            <MULTIPLY type_id="real">
                <VARIABLE id="n" level="2" type_id="integer" />
                <VARIABLE id="y" level="1" type_id="real" />
            </MULTIPLY>
            <VARIABLE id="t" level="2" type_id="real" />
        </SUBTRACT>
    </ASSIGN>
</COMPOUND>


*** FUNCTION func ***


<COMPOUND line="65">
    <ASSIGN line="66" type_id="boolean">
        <VARIABLE id="p" level="2" type_id="boolean" />
        <INTEGER_CONSTANT value="1" type_id="boolean" />
    </ASSIGN>
    <ASSIGN line="67" type_id="boolean">
        <VARIABLE id="q" level="2" type_id="boolean" />
        <INTEGER_CONSTANT value="0" type_id="boolean" />
    </ASSIGN>
    <ASSIGN line="68" type_id="real">
        <VARIABLE id="x" level="2" type_id="real" />
        <ADD type_id="real">
            <SUBTRACT>
                <MULTIPLY type_id="real">
                    <VARIABLE id="i" level="2" type_id="real" />
                    <VARIABLE id="z" level="2" type_id="real" />
                </MULTIPLY>
                <CALL id="func" level="1" type_id="real">
                    <PARAMETERS>
                        <VARIABLE id="v" level="1" type_id="real" />
                        <NEGATE type_id="real">
                            <REAL_CONSTANT value="3.15159" type_id="real" />
                        </NEGATE>
                        <INTEGER_CONSTANT value="5" type_id="integer" />
                    </PARAMETERS>
                </CALL>
            </SUBTRACT>
            <FLOAT_DIVIDE type_id="real">
                <VARIABLE id="n" level="2" type_id="integer" />
                <VARIABLE id="m" level="1" type_id="integer" />
            </FLOAT_DIVIDE>
        </ADD>
    </ASSIGN>
    <ASSIGN line="69" type_id="real">
        <VARIABLE id="func" level="1" type_id="real" />
        <VARIABLE id="x" level="2" type_id="real" />
    </ASSIGN>
</COMPOUND>


*** PROCEDURE nested ***


<COMPOUND line="59">
    <ASSIGN line="60" type_id="subrange">
        <VARIABLE id="s" level="3" type_id="subrange" />
        <VARIABLE id="m" level="3" type_id="integer" />
    </ASSIGN>
    <CALL line="61" id="deeply" level="3" />
    <ASSIGN line="62" type_id="integer">
        <VARIABLE id="a" level="3" type_id="integer" />
        <VARIABLE id="s" level="3" type_id="subrange" />
    </ASSIGN>
</COMPOUND>


*** PROCEDURE deeply ***


<COMPOUND line="53">
    <ASSIGN line="54" type_id="real">
        <VARIABLE id="w" level="4" type_id="real" />
        <VARIABLE id="i" level="2" type_id="real" />
    </ASSIGN>
    <CALL line="55" id="nested" level="2">
        <PARAMETERS>
            <VARIABLE id="a" level="3" type_id="integer" />
            <VARIABLE id="m" level="3" type_id="integer" />
        </PARAMETERS>
    </CALL>
    <ASSIGN line="56" type_id="real">
        <VARIABLE id="w" level="4" type_id="real" />
        <CALL id="forwarded" level="1" type_id="real">
            <PARAMETERS>
                <VARIABLE id="b" level="3" type_id="integer" />
                <VARIABLE id="w" level="4" type_id="real" />
            </PARAMETERS>
        </CALL>
    </ASSIGN>
</COMPOUND>

                    0 instructions generated.
                    0.00 seconds total code generation time.
```

Listing 11-24 shows output with various syntax errors related to declaring and calling procedures and functions.

Listing 11-24: Compilation output with syntax errors

```
001 PROGRAM RoutineErrors (input, output);
002
003 CONST
004     five = 5;
005
006 TYPE
007     enum = (alpha, beta, gamma);
008     arr = ARRAY [1..five] OF real;
```

```
009
010 VAR
011     e, k : enum;
012     i, m : integer;
013     a : arr;
014     v, y : real;
015     t : boolean;
016
017 PROCEDURE proc(j : integer; VAR x, y, z : real;
018                 VAR p : (alpha, beta, gamma); ch : char);
                              ^
*** Invalid type [at "("]
019
020     VAR
021         i : integer;
022
023     BEGIN
024         i := 7*k;
                   ^
*** Incompatible types [at "k"]
025         x := func(x);
                  ^
*** Undefined identifier [at "func"]
                  ^
*** Invalid identifier usage [at "func"]
                  ^
*** Incompatible types [at "func"]
                      ^
*** Unexpected token [at "("]
                        ^
*** Unexpected token [at ")"]
                         ^
*** Missing := [at ";"]
                         ^
*** Unexpected token [at ";"]
                         ^
*** Incompatible types [at ";"]
026     END;
027
028 FUNCTION forwarded(m : integer; VAR t : real) : 1..10; forward;
                                                     ^
*** Invalid type [at "1"]
029
030 FUNCTION func(VAR x : real; i : real; n : integer) : arr;
                                                         ^
*** Invalid type [at ";"]
031
032     VAR
033         z : real;
034         p, q : boolean;
035
036     PROCEDURE nested(VAR n, m : integer);
037
038         VAR
039             a, b, c : integer;
040
041         BEGIN {nested}
042             p := forwarded(9, 2.0);
                                   ^
*** Invalid VAR parameter [at "2.0"]
                  ^
*** Incompatible types [at "forwarded"]
043         END {nested};
044
045     BEGIN {func}
046         x := i*z - func(v, -3.15159, five) + n/m;
                        ^
*** Incompatible types [at "func"]
047         func := x;
                     ^
*** Incompatible types [at "x"]
048     END {func};
049
050 FUNCTION forwarded(b : boolean) : real;
                      ^
*** Already specified in FORWARD [at "("]
051
052     VAR
053         n : integer;
054
055     BEGIN
056         forwarded := true;
                         ^
*** Incompatible types [at "true"]
057     END;
058
059 BEGIN {RoutinesTest}
060     proc(i, -7 + m, a[m], v, y, a, t, &apos;r&apos;);
                ^
*** Invalid VAR parameter [at "-"]
                            ^
*** Invalid VAR parameter [at "y"]
                               ^
*** Incompatible types [at "a"]
                                      ^
*** Wrong number of actual parameters [at "t"]
                                            ^
*** Wrong number of actual parameters [at ";"]
061 END {RoutinesTest}.

                61 source lines.
                23 syntax errors.
               0.13 seconds total parsing time.
```

    This concludes all of the classes in the front end and intermediate tier to generate symbol tables and parse trees that the interpreter back end will need to execute entire Pascal programs. That will be the goal for the next chapter.

Chapter 12

Interpreting Pascal Programs

In Chapters 6 and 8, you wrote executors in the interpreter back end to execute Pascal expressions, assignment statements, and control statements. In the previous chapter, you completed the parser in the front end to parse the program header, procedure and function declarations, and calls to declared and standard procedures and functions. In this chapter, you'll complete the interpreter back end to execute entire Pascal programs.

Goals and Approach

In order to execute entire Pascal program, you must accomplish two major tasks:

- Implement runtime memory management.
- Execute calls to procedures and functions.

You'll first implement memory management. After that's in place, you'll be able to call and execute any procedure or function. Once the interpreter is complete, you can verify it by executing Pascal programs.

Runtime Memory Management

During run time[1], the interpreter manages the memory that the source program uses.

[1] Recall that *run time* is when the interpreter is executing the source program.

[2] Also known as a *stack frame*.

The Runtime Stack and Activation Records

Just as the parser in the front end manages a symbol table stack, the executor in the interpreter back end manages a runtime stack. In the previous chapter, you saw how the parser pushes and pops symbol tables as it enters and exits nested scopes. The interpreter, on the other hand, manages a *runtime stack,* and it pushes and pops *activation records* as it calls procedures and functions and returns from them.

An *activation record*[2] maintains information about the currently executing invocation of a source program routine, which can be a procedure, a function, or the main program itself. In particular, it contains the current values of the routine's formal parameters and local variables.

Note that an activation record represents a particular *invocation* of a routine. If a routine is called recursively, there will be a separate activation record on the runtime stack for each invocation.

By maintaining activation records on the runtime stack, executors can access the local values of the currently executing routine and also the nonlocal values of any enclosing routine.

Memory Maps and Cells

Each activation record contains the current local values for an invocation of a routine. At the conceptual level, consider each activation record to contain a memory map. A memory map contains any number of memory cells. Each cell can hold a value, and it is labeled by the name of a formal parameter or local variable. See Figure 12-1. The cell for an array contains a cell for each array element. The cell for a record value is memory map containing a cell for each field.

Accessing Local Values
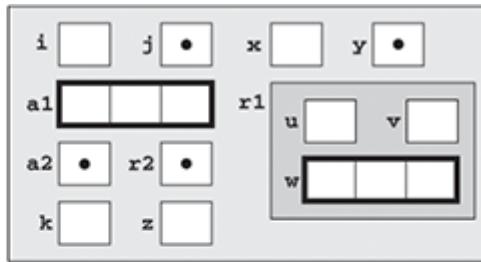
Consider an example program `main1` that contains procedures `proc2a` and `proc2b`. Procedure `proc2a` contains nested procedure `proc3`. (The names of the routines indicate their nesting levels.)

Figure 12-1: Memory maps and memory cells within the activation record for an invocation of procedure `proc`. The cells labeled `i`, `y`, `a2`, and `r2` will contain pointer values, and the rest will contain integer or real values. The cell for `a1` contains a cell for each array element. The cell for `r1` is itself a memory map containing cells for the record fields `u`, `v`, and `w`.

```
TYPE
  arr = ARRAY[1..3] OF integer;
  rec = RECORD
            u, v : real;
            w : arr
          END;
...
PROCEDURE proc(i : integer;
                 x : real;
                 VAR j : integer;
                 VAR y : real;
                 a1 : arr;
                 r1 : rec;
                 VAR a2 : arr,
                 VAR r2 : rec);
  VAR
    k : integer;
    z : real;

  BEGIN {proc}
    ...
  END
```

Design Note

By keeping values in activation records on the runtime stack instead of in symbol tables, you finally remove Hack #5 described at the end of Chapter 6.

There are some key parallels between the symbol table stack and the runtime stack:

- The front end creates symbol tables and manages the symbol table stack as it parses the source program. The parser pushes and pops symbol tables as it enters and exits the nested scopes of procedures and functions.
- The back end creates activation records and manages the runtime stack as it executes the source program. The interpreter pushes and pops activation records as it executes procedure and function calls and returns.

However, there are significant differences between the way the parser searches the symbol table stack and the way the interpreter searches the runtime stack. You'll see below how the interpreter accesses values in the runtime stack.

Figures 12-2a and 12-2b show that accessing local values during run time is simple. The activation record for the currently executing routine's invocation is always on top of the runtime stack.

Figure 12-2a: The executor has pushed activation record *AR:main1* onto the runtime stack for main routine `main1`'s invocation. It finds the values of local variables `i` and `j` in *AR:main1* that is currently at the top of the stack.

```
PROGRAM main1;
VAR i, j : integer;


PROCEDURE proc2a;
  VAR m : integer;


  PROCEDURE proc3;
      VAR j : integer
      BEGIN
        j := i + m;
      END;


  BEGIN {proc2a}
    i := 11;
    m := j;
    proc3;
  END;


PROCEDURE proc2b;
  VAR j : integer;
  BEGIN
    j := 14;
    proc2a;
  END;


BEGIN {main1}
  i := 33;
  j := 55;
  proc2b;
END.
```

Runtime stack

AR: main1
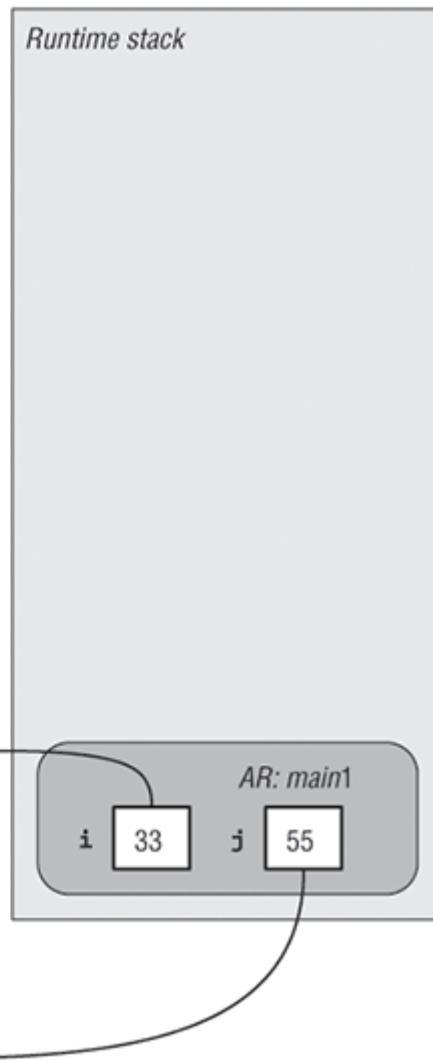
i 33   j 55

Accessing Nonlocal Values

Figures 12-2c and 12-2d show how the executor accesses nonlocal values.

In Figure 12-2c, the executor has pushed activation record *AR:proc2a* onto the stack for proc2a's invocation. It stores the value 11 into nonlocal variable i in the activation record *AR:main1*. The executor also accesses the value of nonlocal variable j from *AR:main1* (not from *AR:proc2b*) which it stores into local variable m in activation record *AR:proc2a* at the top of the stack.

Figure 12-2b: main1 ⇒ proc2b. The executor has pushed activation record *AR:proc2a* onto the stack for proc2b's invocation and finds the value of the procedure's local variable j in *AR:proc2b* that is currently at the top of the stack.

```
PROGRAM main1;
VAR i, j : integer;

PROCEDURE proc2a;
  VAR m : integer;

  PROCEDURE proc3;
      VAR j : integer
      BEGIN
        j := i + m;
      END;

  BEGIN {proc2a}
    i := 11;
    m := j;
    proc3;
  END;

PROCEDURE proc2b;
  VAR j : integer;
  BEGIN
    j := 14;
    proc2a;
  END;

BEGIN {main1}
  i := 33;
  j := 55;
  proc2b;
END.
```

Runtime stack

AR: proc2b
j 14

AR: main1
i 33   j 55
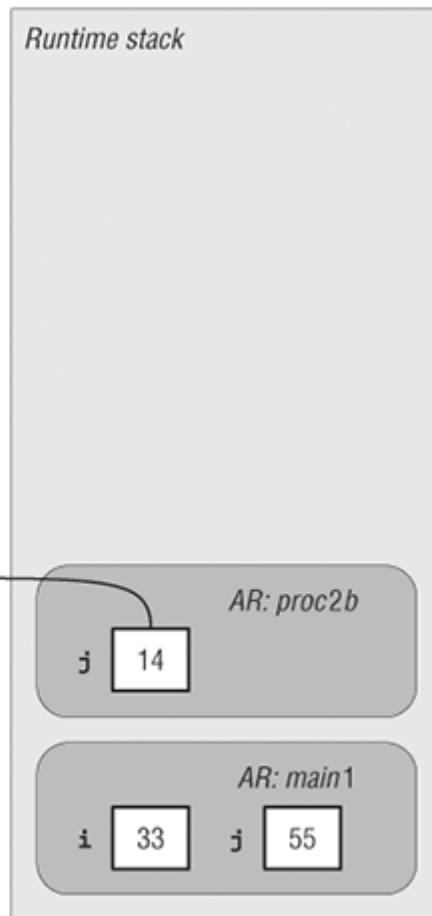
[Figure 12-2c](#) may be a surprise. The executor accesses the value of nonlocal variable `j` not from activation record *AR:proc2b*, but instead from activation record *AR:main1*. This is correct if you examine the Pascal program; the identifier `j` used in the body of procedure `proc2a` is defined in the enclosing scope of `main1`.

Therefore, the back end executor cannot simply search the stacked activation records from top to bottom the way the front end parser searches the stacked symbol tables. At run time, the executor uses the following search algorithm:

- The executor always knows the nesting level of each variable it encounters since that information is stored in the symbol table entry of the variable's identifier.
- If the executor is executing the body of a routine at level n and a variable is also at level n, then the variable must be local to the routine. Otherwise (the nesting level of the variable is less than n), the variable must be nonlocal (defined in an enclosing scope).[3]

    [3] The executor must assume that that the front end parser has done its job correctly and that all variables it encounters in the parse tree are legitimate. In our design, if the parser encountered any syntax errors (such as attempting to access a variable that's out of scope), the back end is never invoked.

    [4] Note that this display has nothing to do with video displays.

- While executing the body of a routine, if the executor needs to access the value of a local variable, that value is stored in the activation record for the routine's current invocation, which is always at the top of the runtime stack.
- While executing the body of a routine at nesting level n, if the executor needs to access the value of a nonlocal variable at level $m < n$, then that value must come from an activation record of a level m routine further down the runtime stack. The executor accesses the nonlocal value in the *topmost* level m activation record.

[Figure 12-2c](#), the executor is executing the body of the level 2 procedure `proc2a`. Variable `j` is defined at level 1 and therefore is nonlocal, so the executor accesses `j`'s value from the topmost level 1 activation record, which is *AR:main1*.

View Figures [12-2a](#) through [12-2d](#) in reverse to see what happens when the executor returns from the routines. Upon returning from `proc3`, the executor pops activation record *AR:proc3* off the stack ([Figure 12-2c](#)). It pops *AR:proc2a* off the stack when it returns

from `proc2a` ([Figure 12-2b](#)), and it pops *AR:proc2b* off the stack when it returns from `proc2b` ([Figure 12-2a](#)). Finally, the executor pops *AR:main1* off the stack when completes the execution of the main program.

The Runtime Display

The back end interpreter can employ a runtime *display* that makes accessing nonlocal values more efficient. The display[4] is an array whose element n points to the topmost activation record on the runtime stack that represents the invocation of a procedure or function at nesting level n. Each activation record also has a *dynamic link* that points to the next activation record at the same level n but lower in the stack if there is one; otherwise, the pointer is null. As you'll see shortly, the executor uses this link to restore the corresponding display element when it executes a return from a routine and pops its activation record off the stack.

Whenever the executor pushes a level n activation record onto the stack, it uses the current pointer value (which can be null) in display element n to set the activation record's dynamic link. In other words, the newly pushed level n activation record points to the level n activation record (if there is one) that was formerly topmost on the stack. The executor then it sets display element n to point to the newly pushed activation record.

Element n of the display provides direct access to the topmost activation record on the stack at nesting level n, thereby improving runtime performance of the executor. Figures [12-3a](#) through [12-3d](#) show how the runtime stack and the runtime display work together.

[Figure 12-2c:](#) `main1` ⟹ `proc2b` ⟹ `proc2a`. The executor has pushed activation record *AR:proc2a* onto the stack for `proc2a`'s invocation. It sets the value of the procedure's local variable `m` in activation record *AR:proc2a* at the top of the stack and the value of nonlocal variable `i` in activation record *AR:main1*. It accesses the value of nonlocal variable `j` in activation record *AR:main1*, not in *AR:proc2b*.
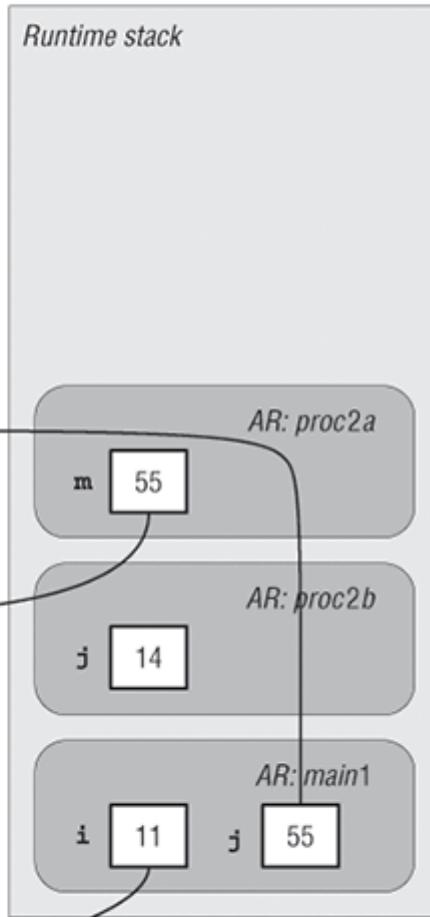

```
PROGRAM main1;

VAR i, j : integer;


PROCEDURE proc2a;

  VAR m : integer;


  PROCEDURE proc3;

      VAR j : integer
      BEGIN

        j := i + m;

      END;


  BEGIN {proc2a}
      i := 11;

      m := j
      proc3;

  END;


PROCEDURE proc2b;

  VAR j : integer;

  BEGIN

      j := 14;

      proc2a;

  END;


BEGIN {main1}

  i := 33;

  j := 55;

  proc2b;

END.
```

[Figure 12-2d:](#) `main1` ⟹ `proc2b` ⟹ `proc2a` ⟹ `proc3`. The executor has pushed activation record *AR:proc3* onto the stack for `proc3`'s invocation. It finds the values of nonlocal variables `i` and `m` in activation records *AR:main1* and *AR:proc2a*, respectively, and stores their sum into the procedure's local variable `j` in activation record *AR:proc3* at the top of the stack.
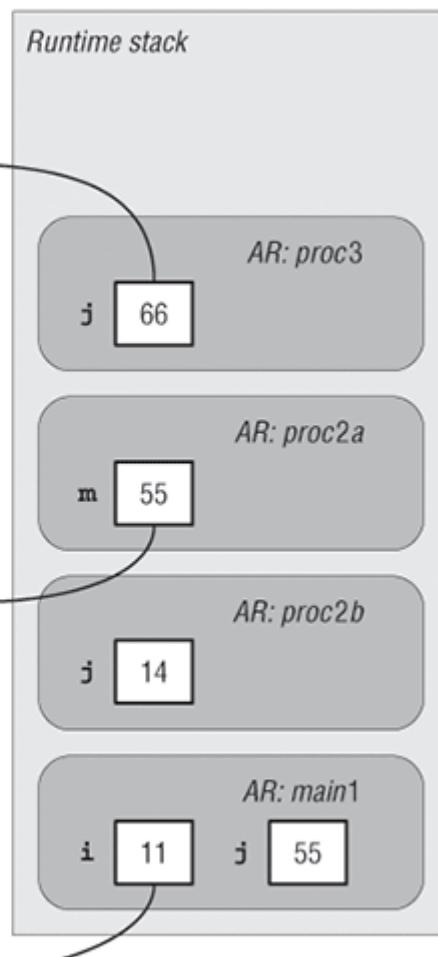
```
PROGRAM main1;
VAR i, j : integer;


PROCEDURE proc2a;
  VAR m : integer;


  PROCEDURE proc3;
     VAR j : integer
     BEGIN
        j := i + m;
     END;


  BEGIN {proc2a}
    i := 11;
    m := j;
    proc3;
  END;


PROCEDURE proc2b;
  VAR j : integer;
  BEGIN
     j := 14;
     proc2a;
  END;


BEGIN {main1}
   i := 33;
   j := 55;
   proc2b;
END.
```


Runtime stack

AR: proc3 — j 66

AR: proc2a — m 55

AR: proc2b — j 14

AR: main1 — i 11, j 55

View these diagrams in reverse order to see what occurs when a routine returns. The executor uses the dynamic link from the activation record it's about to pop off the stack to restore the corresponding display element. So when executing the return from `proc3`, since *AR:proc3* has no dynamic link to another activation record, the executor sets display element 3 to null before popping off *AR:proc3* (from Figure 12-3d to Figure 12-3c). When executing the return from `proc2a`, before popping off *AR:proc2a*, the executor uses *AR:proc2a*'s dynamic link to restore display element 2 to point again at *AR:proc2b* (from Figure 12-3c to Figure 12-3b).

Figure 12-3a: The executor is executing the main program `main1`. It has pushed *AR:main1* onto the stack and set display element 1 to point to it.

```
PROGRAM main1;
VAR i, j : integer;

PROCEDURE proc2a;
  VAR m : integer;

  PROCEDURE proc3;
      VAR j : integer
      BEGIN
        j := i + m;
      END;

  BEGIN {proc2a}
    i := 11;
    m := j;
    proc3;
  END;

PROCEDURE proc2b;
  VAR j : integer;
  BEGIN
    j := 14;
    proc2a;
  END;

BEGIN {main1}
  i := 33;
  j := 55;
  proc2b;
END.
```
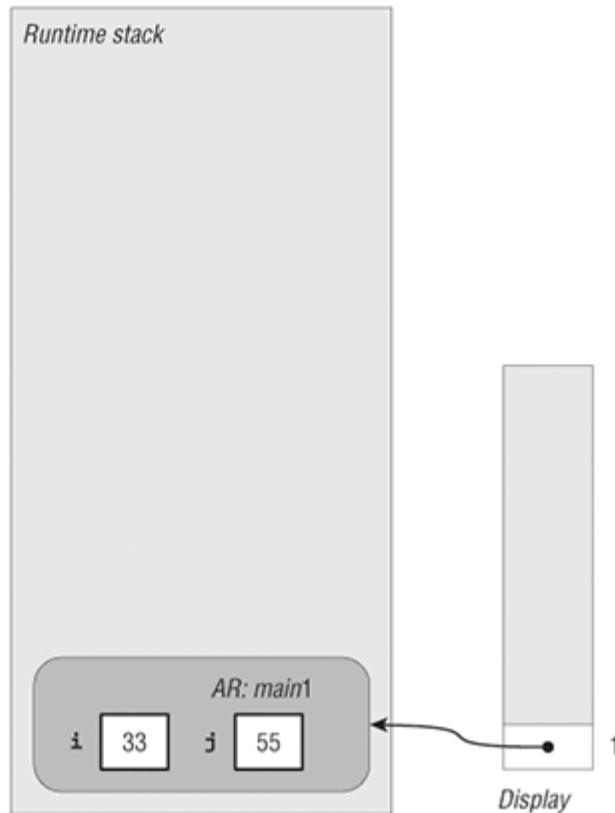

Runtime stack — AR: main1 — i 33 j 55 — Display — 1

Recursive Calls

The runtime stack and activation records enable recursive calls by keeping local values from one invocation of a routine separate from other invocations of the same routine.

Figure 12-4a shows the structure of another example Pascal program. Figures 12-5b through 12-5i show the runtime stack and the runtime display as the program's procedures and functions call each other recursively. In each figure, note what variables the currently executing routine (whose activation record is at the top of the runtime stack) can access.

Figure 12-3b: `main1` ⇒ `proc2b`. The executor has pushed activation record *AR:proc2b* onto the stack and set display element 2 to point to it.

```
PROGRAM main1;
VAR i, j : integer;

PROCEDURE proc2a;
  VAR m : integer;

  PROCEDURE proc3;
      VAR j : integer
      BEGIN
        j := i + m;
      END;

  BEGIN {proc2a}
    i := 11;
    m := j;
    proc3;
  END;

PROCEDURE proc2b;
  VAR j : integer;
  BEGIN
    j := 14;
    proc2a;
  END;

BEGIN {main1}
  i := 33;
  j := 55;
  proc2b;
END.
```
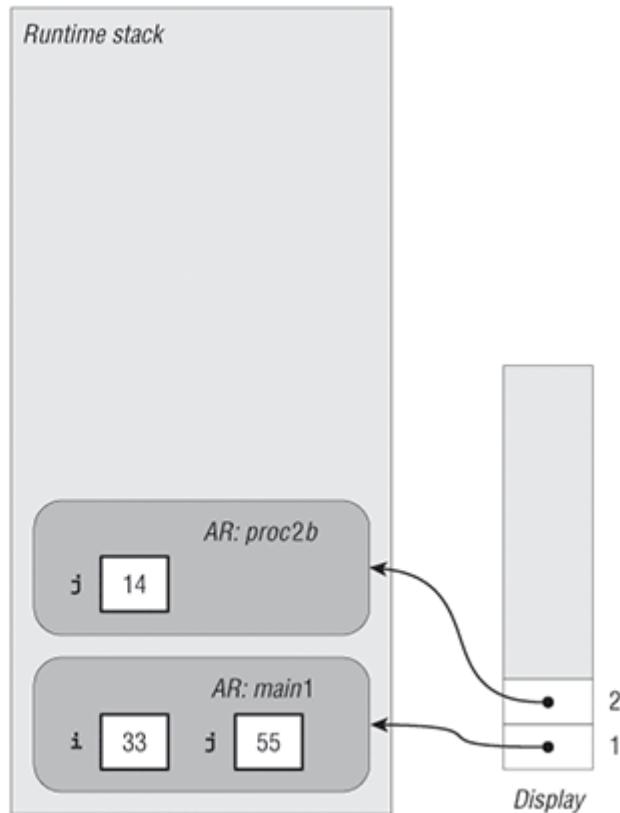

Runtime stack

AR: proc2b
j 14

AR: main1
i 33 j 55

Display
2
1

As before, view the figures in reverse to see the actions of the runtime stack and the runtime display when the routines return.

Memory Management Interfaces and Implementation

Figure 12-5 shows the interfaces that represent the runtime memory components.

Figure 12-3c: `main1` ⇒ `proc2b` ⇒ `proc2a`. The executor has pushed activation record *AR:proc2a* onto the stack. It used the old value of display element 2 (see Figure 12-3b) to set the dynamic link from *AR:proc2a* to *AR:proc2b*, and then it set the display element to point to *AR:proc2a*.

```
PROGRAM main1;
VAR i, j : integer;

PROCEDURE proc2a;
  VAR m : integer;

  PROCEDURE proc3;
      VAR j : integer
      BEGIN
         j := i + m;
      END;

  BEGIN {proc2a}
    i := 11;
    m := j;
    proc3;
  END;

PROCEDURE proc2b;
  VAR j : integer;
  BEGIN
    j := 14;
    proc2a;
  END;

BEGIN {main1}
  i := 33;
  j := 55;
  proc2b;
END.
```
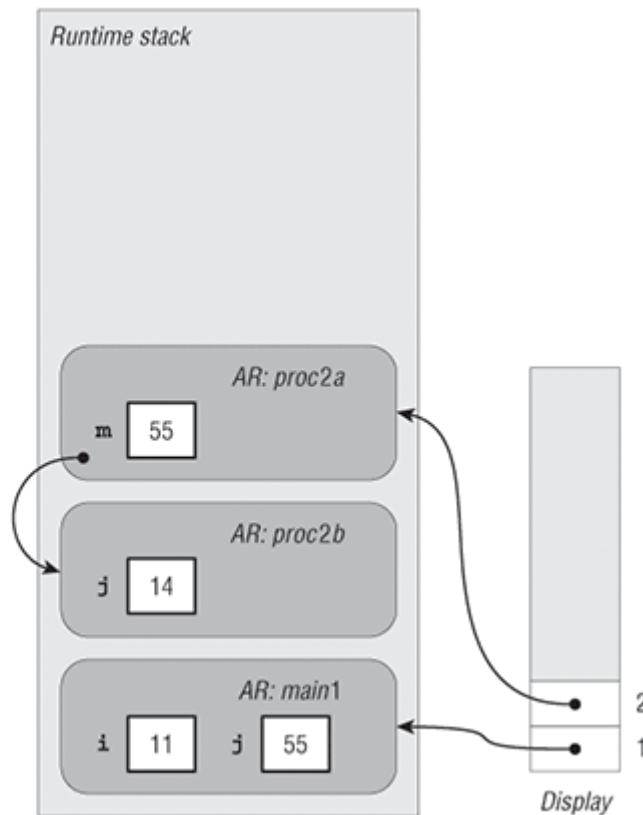
Runtime stack

AR: proc2a

m 55

AR: proc2b

j 14

AR: main1

i 11   j 55

2

1

Display

Figure 12-3d: `main1` ⇒ `proc2b` ⇒ `proc2a` ⇒ `proc3`. The executor has pushed activation record *AR:proc3* onto the stack and set display element 3 to point to it.

```
PROGRAM main1;
VAR i, j : integer;

PROCEDURE proc2a;
  VAR m : integer;

  PROCEDURE proc3;
      VAR j : integer
      BEGIN
         j := i + m;
      END;

  BEGIN {proc2a}
    i := 11;
    m := j;
    proc3;
  END;

PROCEDURE proc2b;
  VAR j : integer;
  BEGIN
    j := 14;
    proc2a;
  END;

BEGIN {main1}
  i := 33;
  j := 55;
  proc2b;
END.
```
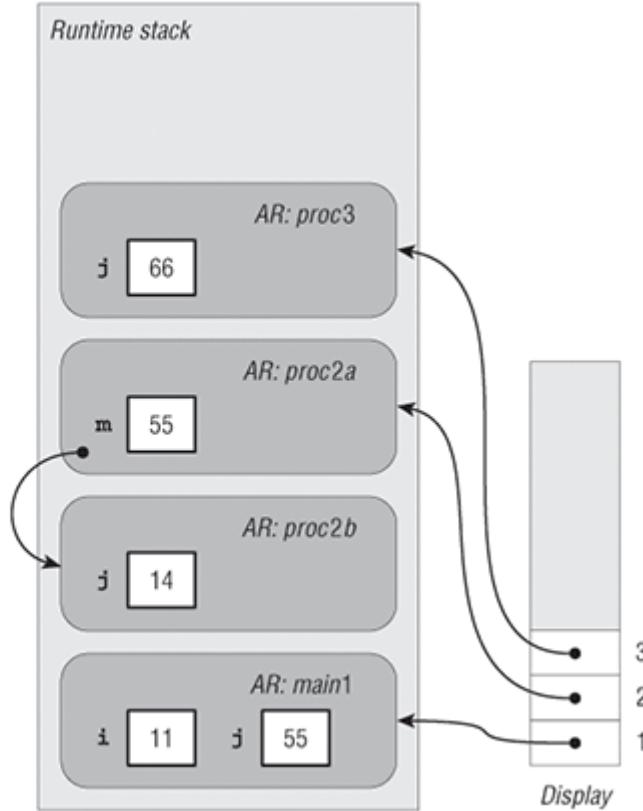
Runtime stack

AR: proc3

j 66

AR: proc2a

m 55

AR: proc2b

j 14

AR: main1

i 11   j 55

3

2

1

Display

Figure 12-4a: The structure of an example Pascal program
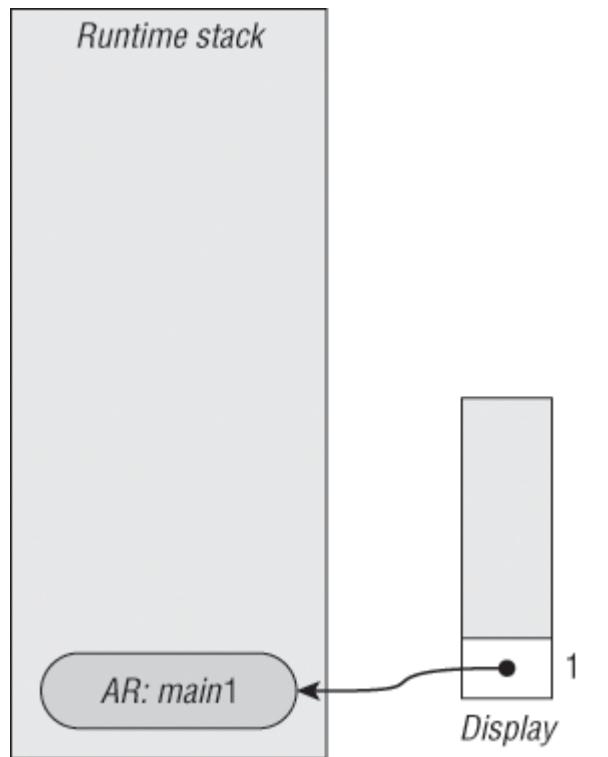
```
PROGRAM main1

    FUNCTION func2

        FUNCTION func3


    PROCEDURE proc2

        PROCEDURE proc3
```
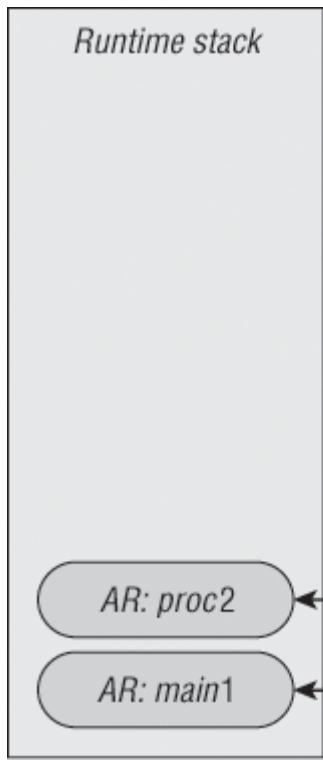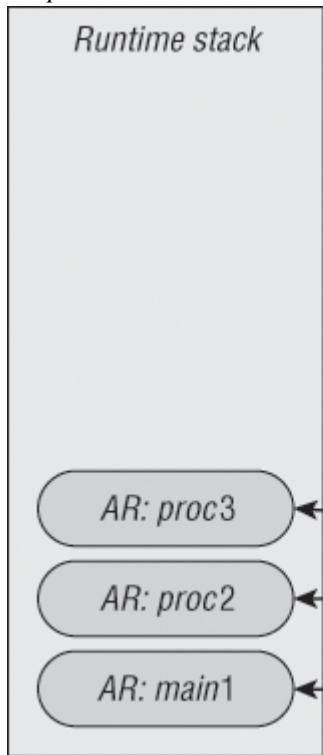
Figure 12-4b: The runtime stack and the display during the execution of the main program `main1`, which can access local values in *AR:main1*

Runtime stack

AR: main1

1

Display
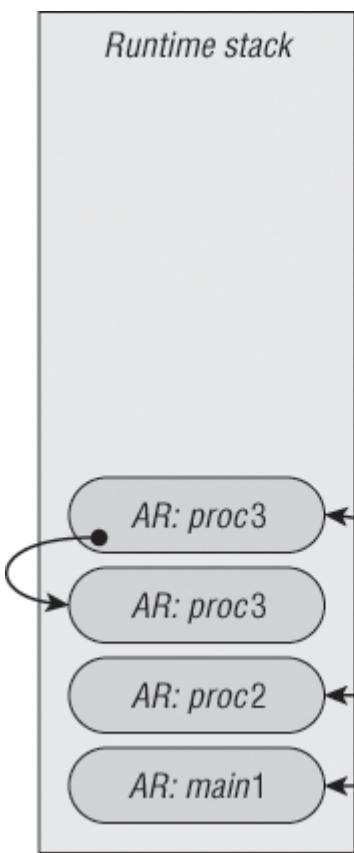
Figure 12-4c: `main1` ⇒ `proc2`. This invocation of `proc2` can access local values in *AR:proc2* and nonlocal values from *AR:main1*.
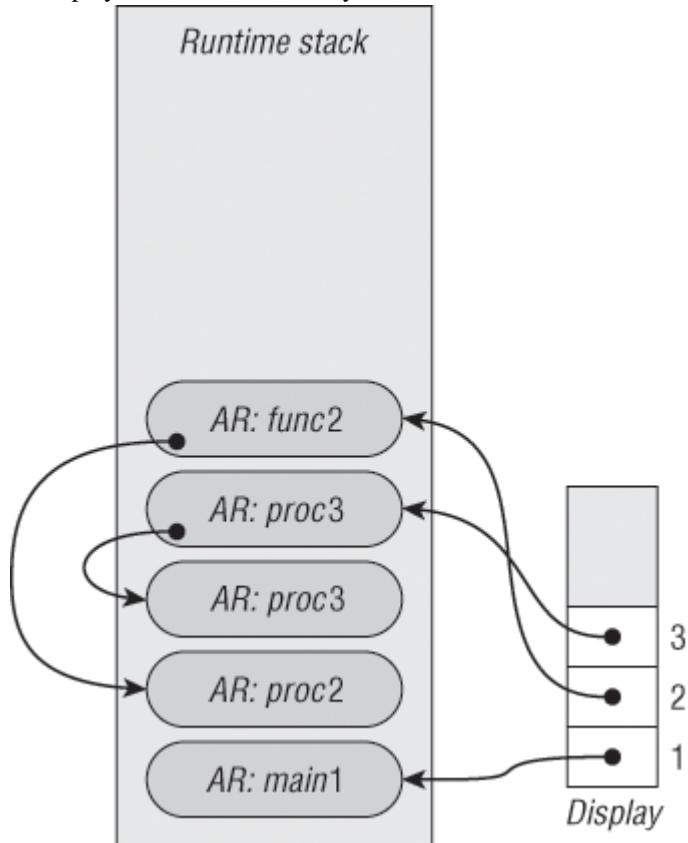
Figure 12-4d: `main1` ⇒ `proc2` ⇒ `proc3`. This invocation of `proc3` can access local values in *AR:proc3* and nonlocal values in *AR:proc2* and *AR:main1*.
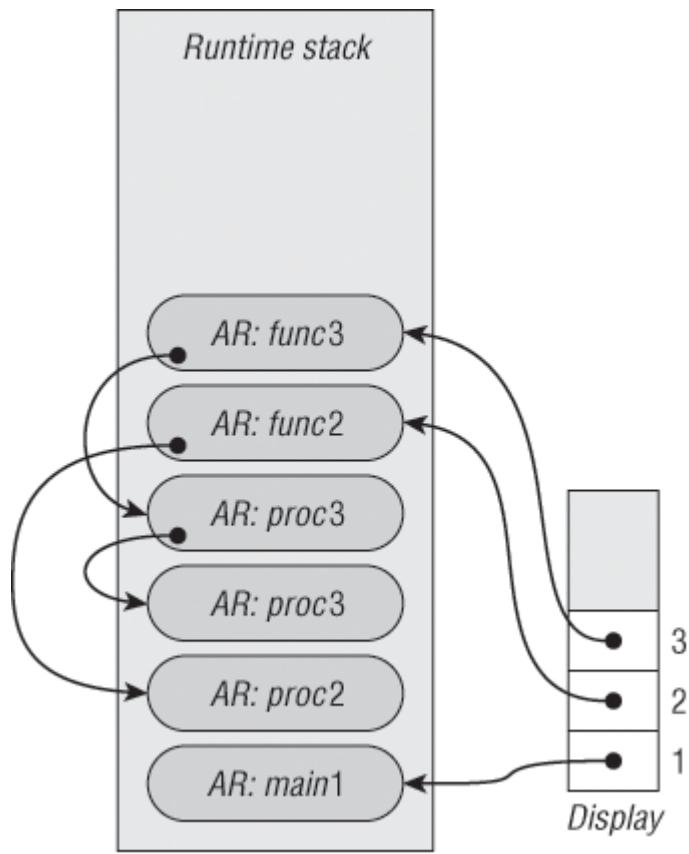
Figure 12-4e: `main1` ⇒ `proc2` ⇒ `proc3` ⇒ `proc3`. After the recursive call, the second invocation of `proc3` can access local values in the topmost *AR:proc3* and nonlocal values in *AR:proc2* and *AR:main1*. It cannot access values inside the lower *AR:proc3*. The activation records keep the local values of `proc3`'s invocations separate from each other.
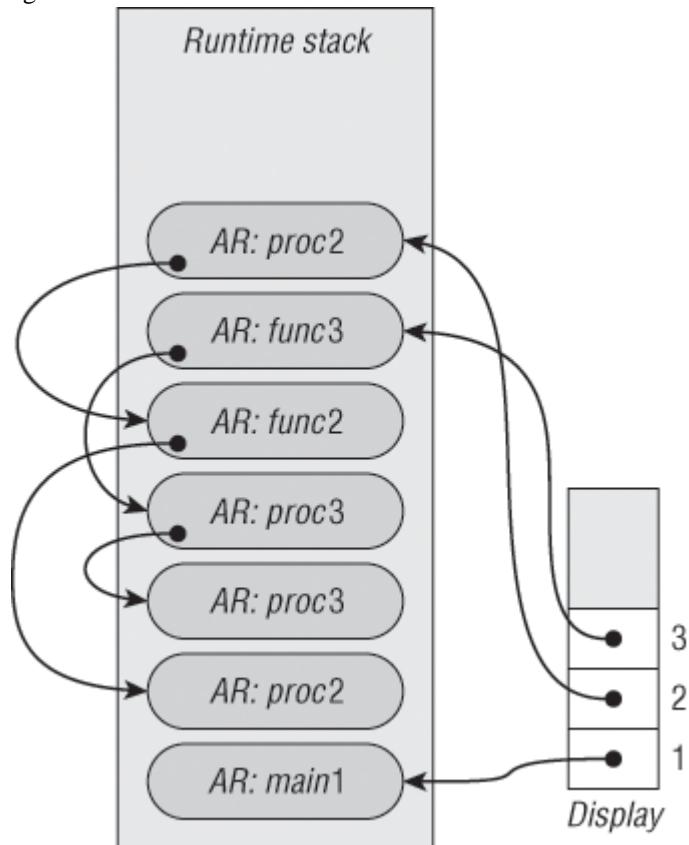
Figure 12-4f: `main1` ⇒ `proc2` ⇒ `proc3` ⇒ `proc3` ⇒ `func2`. This invocation of `func2` can access local values in *AR:func2* and nonlocal values *AR:main1*. Since the front end parser did its job correctly, `func2` can contain no references to any variables declared in `proc3` so display element 3 is currently not used.

Figure 12-4g: `main1` ⇒ `proc2` ⇒ `proc3` ⇒ `proc3` ⇒ `func2` ⇒ `func3`. This invocation of `func3` can access local values in *AR:func3* and nonlocal values in *AR:func2* and *AR:main1*.
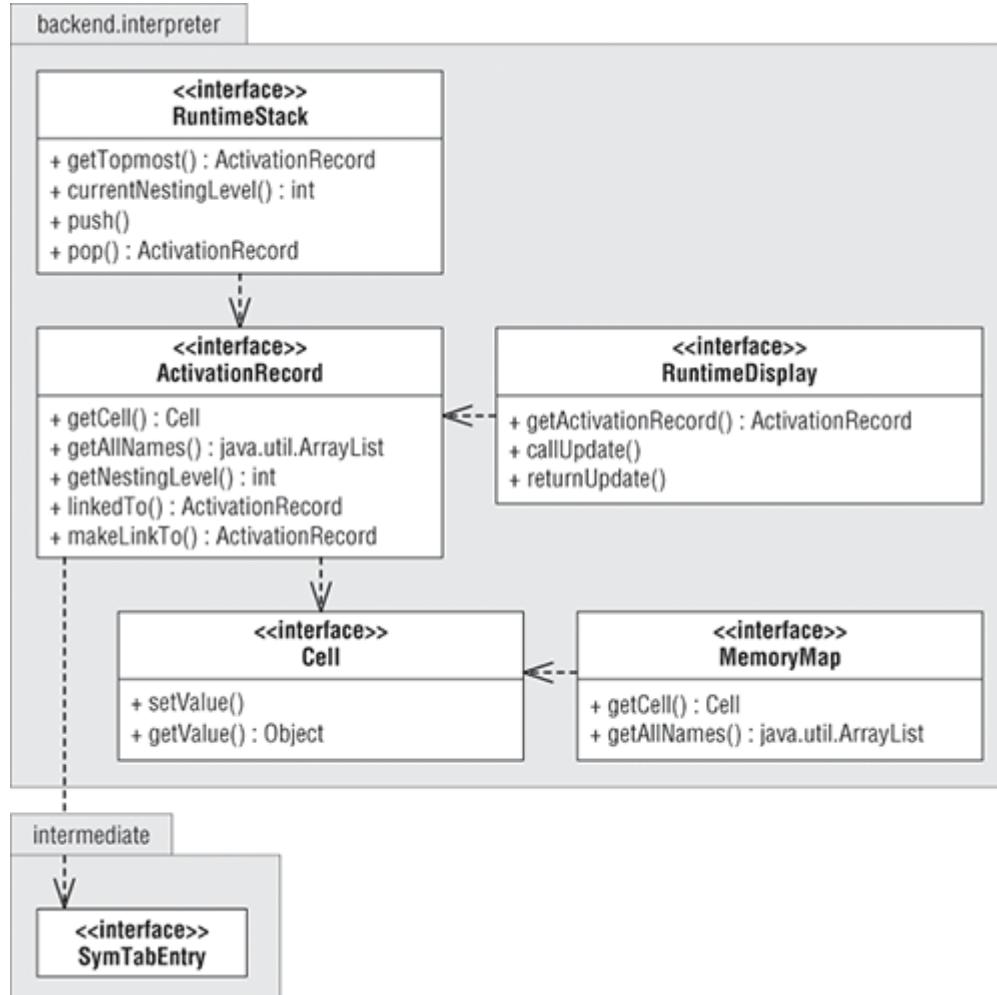
Figure 12-4h: `main1` ⇒ `proc2` ⇒ `proc3` ⇒ `proc3` ⇒ `func2` ⇒ `func3` ⇒ `proc2`. This invocation of `proc2` can access local values in *AR:proc2* and nonlocal values in *AR:main1*. `proc2` can contain no references to any variables declared in `func3`, so display element 3 again is not used.

`main1` ⇒ `proc2` ⇒ `proc3` ⇒ `proc3` ⇒ `func2` ⇒ `func3` ⇒ `proc2` ⇒ `proc3`. This invocation of `proc3` can access local values in *AR:proc3* and nonlocal values in *AR:proc2* and *AR:main1*.

Figure 12-5: The runtime memory components

Both interfaces `RuntimeStack` and `RuntimeDisplay` depend on interface `ActivationRecord`. Interface `ActivationRecord` depends on interface `SymTabEntry` which will represent the name of the procedure or function whose invocation generated the activation record. Both interfaces `ActivationRecord` and `MemoryMap` depend on interface `Cell`.

All the interfaces shown in Figure 12-5 are defined in package `backend.interpreter`, and their implementations are defined in package `backend.interpreter.memoryimpl`. The implementation classes are mostly straightforward.

Interface `RuntimeStack` defines method `push()` to push a routine's activation record onto the stack when the routine is called and method `pop()` to pop off and return an activation record when a routine returns. Method `records()` returns an array list of the activation records on the stack, and method `getTopmost()` returns the topmost activation record from the stack at a given nesting level. Method `currentNestingLevel()` returns the current scope nesting level. shows this interface.

Listing 12-1: Interface `RuntimeStack`

```
/**
 * <h1>RuntimeStack</h1>
 *
 * <p>Interface for the interpreter&apos;s runtime stack.</p>
 */
public interface RuntimeStack
{
    /**
     * @return an array list of the activation records on the stack.
     */
    public ArrayList<ActivationRecord> records();

    /**
     * Get the topmost activation record at a given nesting level.
     * @param nestingLevel the nesting level.
     * @return the activation record.
     */
    public ActivationRecord getTopmost(int nestingLevel);

    /**
     * @return the current nesting level.
     */
    public int currentNestingLevel();

    /**
     * Pop an activation record off the stack.
     */

    public void pop();
    /**
     * Push an activation record onto the stack.
     * @param ar the activation record to push.
     */
    public void push(ActivationRecord ar);
}
```

shows the methods of implementation class `RuntimeStackImpl`. Implement the stack as an array list.

Listing 12-2: Class `RuntimeStackImpl`

```
/**
 * <h1>RuntimeStackImpl</h1>
 *
 * <p>The interpreter&apos;s runtime stack.</p>
 */
public class RuntimeStackImpl
    extends ArrayList<ActivationRecord>
    implements RuntimeStack
{
    private RuntimeDisplay display;  // runtime display

    /**
     * Constructor.
     */
    public RuntimeStackImpl()
    {
        display = MemoryFactory.createRuntimeDisplay();
    }

    /**
     * @return an array list of the activation records on the stack.
     */
    public ArrayList<ActivationRecord> records()
    {
        return this;
    }

    /**
     * Get the topmost activation record at a given nesting level.
     * @param nestingLevel the nesting level.
     * @return the activation record.
     */
    public ActivationRecord getTopmost(int nestingLevel)
    {
        return display.getActivationRecord(nestingLevel);
    }

    /**
     * @return the current nesting level.
     */
    public int currentNestingLevel()
    {
        int topIndex = size() - 1;
        return topIndex >= 0 ? get(topIndex).getNestingLevel() : -1;
    }

    /**
     * Push an activation record onto the stack for a routine being called.
     * @param ar the activation record to push.
     */
    public void push(ActivationRecord ar)
    {
        int nestingLevel = ar.getNestingLevel();

        add(ar);
        display.callUpdate(nestingLevel, ar);
    }

    /**
     * Pop an activation record off the stack for a returning routine.
     */
    public void pop()
    {
        display.returnUpdate(currentNestingLevel());
        remove(size() - 1);
```

```
        }
    }
```

Each runtime stack owns a runtime display that the constructor creates. Method `getTopmost()` uses the display to access the desired activation record. Method `currentNestingLevel()` returns the current nesting level which it obtains from the activation record at the top of the runtime stack. Method `push()` adds an activation record at the top end of the array list and calls `display.callUpdate()` to update the display. Method `pop()` calls `display.returnUpdate()` to update the display before removing the activation record from the top end of the array list.

[Listing 12-3](#) shows interface `RuntimeDisplay`. Method `getActivationRecord()` returns the activation record pointed to by the element for a given nesting level. This is, of course, the topmost activation record on the stack at that level. Method `callUpdate()` updates the display for a given nesting level whenever the executor executes a call to a procedure or function. It is passed the routine's activation record. Method `returnUpdate()` updates the display for a given nesting level whenever the executor executes a return from a procedure or function.

[Listing 12-3:](#) Interface `RuntimeDisplay`

```
/**
 * <h1>RuntimeDisplay</h1>
 *
 * <p>Interface for the interpreter&apos;s runtime display.</p>
 */
public interface RuntimeDisplay
{
    /**
     * Get the activation record at a given nesting level.
     * @param nestingLevel the nesting level.
     * @return the activation record.
     */
    public ActivationRecord getActivationRecord(int nestingLevel);

    /**
     * Update the display for a call to a routine at a given nesting level.
     * @param nestingLevel the nesting level.
     * @param ar the activation record for the routine.
     */
    public void callUpdate(int nestingLevel, ActivationRecord ar);

    /**
     * Update the display for a return from a routine at a given nesting level.
     * @param nestingLevel the nesting level.
     */
    public void returnUpdate(int nestingLevel);
}
```

[Listing 12-4](#) shows the methods of implementation class `RuntimeDisplayImpl`, which you also implement as an array list.

[Listing 12-4:](#) Class `RuntimeDisplayImpl`

```
/**
 * <h1>RuntimeDisplayImpl</h1>
 *
 * <p>The interpreter&apos;s runtime display.</p>
 */
public class RuntimeDisplayImpl
    extends ArrayList<ActivationRecord>
    implements RuntimeDisplay
{
    /**
     * Constructor.
     */
    public RuntimeDisplayImpl()
    {
        add(null);  // dummy element 0 (never used)
    }

    /**
     * Get the activation record at a given nesting level.
     * @param nestingLevel the nesting level.
     * @return the activation record.
     */
    public ActivationRecord getActivationRecord(int nestingLevel)
    {
        return get(nestingLevel);
    }

    /**
     * Update the display for a call to a routine at a given nesting level.
     * @param nestingLevel the nesting level.
     * @param ar the activation record for the routine.
     */
    public void callUpdate(int nestingLevel, ActivationRecord ar)
    {
        // Next higher nesting level: Append a new element at the top.
        if (nestingLevel >= size()) {
            add(ar);
        }

        // Existing nesting level: Set at the specified level.
        else {
            ActivationRecord prevAr = get(nestingLevel);
            set(nestingLevel, ar.makeLinkTo(prevAr));
        }
    }

    /**
     * Update the display for a return from a routine at a given nesting level.
     * @param nestingLevel the nesting level.
     */
    public void returnUpdate(int nestingLevel)
    {
        int topIndex = size() - 1;
        ActivationRecord ar = get(nestingLevel);  // AR about to be popped off
        ActivationRecord prevAr = ar.linkedTo();  // previous AR it points to

        // Point the element at that nesting level to the
        // previous activation record.
        if (prevAr != null) {
            set(nestingLevel, prevAr);
        }

        // The top element has become null, so remove it.
```

```
            else if (nestingLevel == topIndex) {
                remove(topIndex);
            }
        }
    }
}
```

The constructor creates a dummy element 0 which will never be used, since there are no variables at nesting level 0 (global). Doing so will make it easier to access the other elements using the nesting level as an index.

Method `callUpdate()` either increases the size of display by appending a new element at the top end (as in going from Figure 12-4c to Figure 12-4d), or it updates an existing element (from Figure 12-4d to Figure 12-4e). In the latter case, the method also sets the dynamic link of the activation record of the routine being called.

When a routine returns, method `returnUpdate()` updates the display element for the routine's nesting level by using the dynamic link of the routine's activation record (from Figure 12-4e to Figure 12-4d). However, if the dynamic link is null and it's the topmost display element, the method simply removes that element and shrinks the size of the display (from Figure 12-4d to Figure 12-4c).

Listing 12-5 shows interface `ActivationRecord`.

Listing 12-5: Interface `ActivationRecord`

```
/**
 * <h1>ActivationRecord</h1>
 *
 * <p>Interface for the interpreter&apos;s runtime activation record.</p>
 */
public interface ActivationRecord
{
    /**
     * Getter.
     * @return the symbol table entry of the routine&apos;s name.
     */
    public SymTabEntry getRoutineId();

    /**
     * Return the memory cell for the given name from the memory map.
     * @param name the name.
     * @return the cell.
     */
    public Cell getCell(String name);

    /**
     * @return the list of all the names in the memory map.
     */
    public ArrayList<String> getAllNames();

    /**
     * Getter.
     * @return the scope nesting level.
     */
    public int getNestingLevel();

    /**
     * @return the activation record to which this record is dynamically linked.
     */
    public ActivationRecord linkedTo();

    /**
     * Make a dynamic link from this activation record to another one.
     * @param ar the activation record to link to.
     * @return this activation record.
     */
    public ActivationRecord makeLinkTo(ActivationRecord ar);
}
```

Getter method `getRoutineId()` returns the symbol table entry of the name of the procedure or function whose invocation generated this activation record. Method `getCell()` returns the memory cell with a given label name. Method `getAllNames()` returns an array list of all the names in the activation record's memory map. Method `getNestingLevel()` returns the nesting level of the routine to which this activation record belongs. Method `linkedTo()` returns the activation record at the same nesting level but lower in the runtime stack pointed to by the dynamic link. Method `makeLinkTo()` sets the dynamic link.

Listing 12-6 shows the key methods of implementation class `ActivationRecordImpl`.

Listing 12-6: Class `ActivationRecord`

```
/**
 * <h1>ActivationRecordImpl</h1>
 *
 * <p>The runtime activation record.</p>
 */
public class ActivationRecordImpl implements ActivationRecord
{
    private SymTabEntry routineId;  // symbol table entry of the routine&apos;s name
    private ActivationRecord link;  // dynamic link to the previous record
    private int nestingLevel;       // scope nesting level of this record
    private MemoryMap memoryMap;    // memory map of this stack record

    /**
     * Constructor.
     * @param routineId the symbol table entry of the routine&apos;s name.
     */
    public ActivationRecordImpl(SymTabEntry routineId)
    {
        SymTab symTab = (SymTab) routineId.getAttribute(ROUTINE_SYMTAB);

        this.routineId = routineId;
        this.nestingLevel = symTab.getNestingLevel();
        this.memoryMap = MemoryFactory.createMemoryMap(symTab);
    }

    /**
     * Getter.
     * @return the symbol table entry of the routine&apos;s name.
     */
    public SymTabEntry getRoutineId()
    {
        return routineId;
    }

    /**
     * Return the memory cell for the given name from the memory map.
     * @param name the name.
```

```java
     * @return the cell.
     */
    public Cell getCell(String name)
    {
        return memoryMap.getCell(name);
    }

    /**
     * @return the list of all the names in the memory map.
     */
    public ArrayList<String> getAllNames()
    {
        return memoryMap.getAllNames();
    }

    /**
     * Getter.
     * @return the scope nesting level.
     */
    public int getNestingLevel()
    {
        return nestingLevel;
    }

    /**
     * @return the activation record to which this record is dynamically linked.
     */
    public ActivationRecord linkedTo()
    {
        return link;
    }

    /**
     * Make a dynamic link from this activation record to another one.
     * @param ar the activation record to link to.
     * @return this activation record.
     */
    public ActivationRecord makeLinkTo(ActivationRecord ar)
    {
        link = ar;
        return this;
    }
}
```

The constructor creates the activation record's memory map. It uses the symbol table entry of the invoked routine's name to obtain the routine's symbol table which will determine how to allocate the memory map's cells. It also obtains the nesting level from the symbol table.

Method `getCell()` returns the memory cell with the given name. Method `getAllNames()` retrieves the array list of all the names in the memory map. Method `linkedTo()` returns the activation record's dynamic link. Method `makeLinkTo()` sets the dynamic link to a given activation record, which the method returns.

Listing 12-7 shows interface `MemoryMap`.

Listing 12-7: Interface `MemoryMap`

```java
/**
 * <h1>MemoryMap</h1>
 *
 * <p>Interface for the interpreter&apos;s runtime memory map.</p>
 */
public interface MemoryMap
{
    /**
     * Return the memory cell with the given name.
     * @param name the name.
     * @return the cell.
     */
    public Cell getCell(String name);

    /**
     * @return the list of all the names.
     */
    public ArrayList<String> getAllNames();
}
```

Method `getCell()`returns the memory cell with the given name, and method `getAllNames()` returns an array list of all the names in the memory map.

Implementation class `MemoryMapImpl`, shown in Listing 12-8, has the most work of all the runtime memory classes. Implement it as a hash map.

Listing 12-8: Class `MemoryMapImpl`

```java
/**
 * <h1>MemoryMapImpl</h1>
 *
 * <p>The interpreter&apos;s runtime memory map.</p>
 *
 */
public class MemoryMapImpl
    extends HashMap<String, Cell>
    implements MemoryMap
{
    /**
     * Constructor.
     * Create a memory map and allocate its memory cells
     * based on the entries in a symbol table.
     * @param symTab the symbol table.
     */
    public MemoryMapImpl(SymTab symTab)
    {
        ArrayList<SymTabEntry> entries = symTab.sortedEntries();

        // Loop for each entry of the symbol table.
        for (SymTabEntry entry : entries) {
            Definition defn = entry.getDefinition();

            // Not a VAR parameter: Allocate cells for the data type
            //                      in the hashmap.
            if ((defn == VARIABLE) || (defn == FUNCTION) ||
                (defn == VALUE_PARM) || (defn == FIELD))
```

```
                {
                    String name = entry.getName();
                    TypeSpec type = entry.getTypeSpec();
                    put(name, MemoryFactory.createCell(allocateCellValue(type)));
                }

                // VAR parameter: Allocate a single cell to hold a reference
                //                 in the hashmap.
                else if (defn == VAR_PARM) {
                    String name = entry.getName();
                    put(name, MemoryFactory.createCell(null));
                }
            }
        }

    /**
     * Return the memory cell with the given name.
     * @param name the name.
     * @return the cell.
     */
    public Cell getCell(String name)
    {
        return get(name);
    }

    /**
     * @return the list of all the names.
     */
    public ArrayList<String> getAllNames()
    {
        ArrayList<String> list = new ArrayList<String>();

        Set<String> names = keySet();
        Iterator<String> it = names.iterator();

        while (it.hasNext()) {
            list.add(it.next());
        }

        return list;
    }

    /**
     * Make an allocation for a value of a given data type for a memory cell.
     * @param type the data type.
     * @return the allocation.
     */
    private Object allocateCellValue(TypeSpec type)
    {
        TypeForm form = type.getForm();

        switch ((TypeFormImpl) form) {

            case ARRAY: {
                return allocateArrayCells(type);
            }

            case RECORD: {
                return allocateRecordMap(type);
            }

            default: {
                return null;  // uninitialized scalar value
            }
        }
    }

    /**
     * Allocate the memory cells of an array.
     * @param type the array type.
     * @return the allocation.
     */
    private Object[] allocateArrayCells(TypeSpec type)
    {
        int elmtCount = (Integer) type.getAttribute(ARRAY_ELEMENT_COUNT);
        TypeSpec elmtType = (TypeSpec) type.getAttribute(ARRAY_ELEMENT_TYPE);
        Cell allocation[] = new Cell[elmtCount];

        for (int i = 0; i < elmtCount; ++i) {
            allocation[i] =
                MemoryFactory.createCell(allocateCellValue(elmtType));
        }

        return allocation;
    }

    /**
     * Allocate the memory map for a record.
     * @param type the record type.
     * @return the allocation.
     */
    private MemoryMap allocateRecordMap(TypeSpec type)
    {
        SymTab symTab = (SymTab) type.getAttribute(RECORD_SYMTAB);
        MemoryMap memoryMap = MemoryFactory.createMemoryMap(symTab);

        return memoryMap;
    }
}
```

The constructor allocates the memory map's cells based on the entries in the symbol table of the procedure or function being called (or of the main program). For a variable, function return value, value parameter, or a record field, it calls `allocateCellValue()` to allocate the appropriate number of cells for the value. For a VAR parameter, it creates a single cell to hold a reference pointer. Each cell is labeled with the name from the symbol table entry.

Method `getCell()` returns the cell with the given name and method `getAllNames()` returns an array list of all the names.

Private methods `allocateCellValue()`, `allocateArrayCells()`, and `allocateRecordMap()` do the cell allocation work. Method `allocateCellValue()` calls `allocateArrayCells()` for an array value and `allocateRecordMap()` for a record value.

Method `allocateArrayCells()` creates a cell for each array element. The result is a memory cell that contains multiple memory cells.

Method `allocateRecordMap()` creates a memory map that contains cells for the record fields. The result is a memory cell that contains the record value's memory map.

Listing 12-9 shows interface `Cell`. It defines two methods: `setValue()`, to set the cell's value, and `getValue()` to return the value.

Listing 12-9: Interface `Cell`

```
/**
 * <h1>Cell</h1>
 *
 * <p>Interface for the interpreter&apos;s runtime memory cell.</p>
 */
public interface Cell
{
    /**
     * Set a new value into the cell.
     * @param newValue the new value.
     */
    public void setValue(Object newValue);

    /**
     * @return the value in the cell.
     */
    public Object getValue();
}
```

Listing 12-10 shows the methods of implementation class `CellImpl`.

Listing 12-10: Class `CellImpl`

```
/**
 * <h1>CellImpl</h1>
 *
 * <p>The interpreter&apos;s runtime memory cell.</p>
 */
public class CellImpl implements Cell
{
    private Object value = null;  // value contained in the memory cell

    /**
     * Constructor
     * @param value the value for the cell.
     */
    public CellImpl(Object value)
    {
        this.value = value;
    }

    /**
     * Set a new value into the cell.
     * @param newValue the new value.
     */
    public void setValue(Object newValue)
    {
        value = newValue;
    }

    /**
     * @return the value in the cell.
     */
    public Object getValue()
    {
        return value;
    }
}
```

The private field `value` stores the cell's value, which method `setValue()` sets and method `getValue()` returns.

Design Note

Create `Cell` objects rather than simply using Java's native objects such as `Integer` and `Float`. Java's objects are immutable; once you've created one to wrap a scalar value, you cannot change that value. On the other hand, a `Cell` object provides a reference to an object whose value can change.

The Memory Factory

Listing 12-11 shows the static methods of class `MemoryFactory` that create the various runtime memory management objects.

Design Note

As with all the previous factory classes, class `MemoryFactory` allows you to decouple the conceptual design of runtime memory management from its implementation.

Listing 12-11: Class `MemoryFactory`

```
/**
 * <h1>MemoryFactory</h1>
 *
 * <p>A factory class that creates runtime components.</p>
 */
public class MemoryFactory
{
    /**
     * Create a runtime stack.
     * @return the new runtime stack.
     */
    public static RuntimeStack createRuntimeStack()
    {
        return new RuntimeStackImpl();
    }

    /**
```

```
     * Create a runtime display.
     * @return the new runtime display.
     */
    public static RuntimeDisplay createRuntimeDisplay()
    {
        return new RuntimeDisplayImpl();
    }

    /**
     * Create an activation record for a routine.
     * @param routineId the symbol table entry of the routine&apos;s name.
     * @return the new activation record.
     */
    public static ActivationRecord createActivationRecord(SymTabEntry routineId)
    {
        return new ActivationRecordImpl(routineId);
    }

    /**
     * Create a memory map from a symbol table.
     * @param value the value for the cell.
     * @return the new memory map.
     */
    public static MemoryMap createMemoryMap(SymTab symTab)
    {
        return new MemoryMapImpl(symTab);
    }

    /**
     * Create a memory cell.
     * @param value the value for the cell.
     * @return the new memory cell.
     */
    public static Cell createCell(Object value)
    {
        return new CellImpl(value);
    }
}
```

Executing Statements and Expressions

Now that you've implemented runtime memory management, complete some of the back end interpreter work that you began in Chapters 6 and 8.

The Executor Superclass

Listing 12-12 shows new versions of the static initializer and the `process()` method of class `Executor` in package `backend.interpreter`, the superclass of all the executor subclasses. (Review Figure 6-1.)

Listing 12-12: The static initializer and the `process()` method of class `Executor`

```
public class Executor extends Backend
{
    protected static int executionCount;
    protected static RuntimeStack runtimeStack;
    protected static RuntimeErrorHandler errorHandler;

    protected static Scanner standardIn;       // standard input
    protected static PrintWriter standardOut;  // standard output

    static {
        executionCount = 0;
        runtimeStack = MemoryFactory.createRuntimeStack();
        errorHandler = new RuntimeErrorHandler();

        try {
            standardIn = new PascalScanner(
                            new Source(
                                new BufferedReader(
                                    new InputStreamReader(System.in))));
            standardOut = new PrintWriter(
                              new PrintStream(System.out));
        }
        catch (IOException ignored) {
        }
    }

    /**
     * Execute the source program by processing the intermediate code
     * and the symbol table stack generated by the parser.
     * @param iCode the intermediate code.
     * @param symTabStack the symbol table stack.
     * @throws Exception if an error occurred.
     */
    public void process(ICode iCode, SymTabStack symTabStack)
        throws Exception
    {
        this.symTabStack = symTabStack;
        long startTime = System.currentTimeMillis();

        SymTabEntry programId = symTabStack.getProgramId();

        // Construct an artificial CALL node to the main program.
        ICodeNode callNode = ICodeFactory.createICodeNode(CALL);
        callNode.setAttribute(ID, programId);

        // Execute the main program.
        CallDeclaredExecutor callExecutor = new CallDeclaredExecutor(this);
        callExecutor.execute(callNode);

        float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
        int runtimeErrors = errorHandler.getErrorCount();

        // Send the interpreter summary message.
        sendMessage(new Message(INTERPRETER_SUMMARY,
                        new Number[] {executionCount,
                                      runtimeErrors,
                                      elapsedTime}));
    }
}
```

Class `Executor` now has three additional static fields: `runtimeStack`, `standardIn`, and `standardOut`. The static initializer uses the memory factory to create the runtime stack. It sets `standardIn` to be a new instance of class `PascalScanner` that is associated with a new instance of class `Source` that is ultimately based on `System.in`. The initializer also sets `standardOut` to be based on `System.out`. These will be used by the executing source program.

Design Note

By basing the runtime standard input on the front end classes `PascalScanner` and Source, you can reuse the functionality of these two classes at runtime whenever the source program reads input text. You'll soon see that you can implement the standard Pascal procedures `read` and `readln` as rudimentary scanners of the input text.

The `process()` method constructs an artificial `CALL` parse tree node in order to execute a "call" to the main program via `callExecutor.execute()`. You'll examine executing calls to procedures and functions (and to the main program) shortly.

The Statement Executor

Class `StatementExecutor` changes substantially to accommodate runtime memory management and to provide debugging and tracing feedback while executing the source program.

First, add the following `case` to the `switch` statement in method `execute()` to handle a call to a procedure or function:

```
case CALL: {
    CallExecutor callExecutor = new CallExecutor(this);
    return callExecutor.execute(node);
}
```

[Listing 12-13](#) shows a number of new class `StatementExecutor` methods that the statement executor subclasses need to handle values during run time.

[Listing 12-13:](#) Methods of class `StatementExecutor` that handle values during run time

```java
/**
 * Convert a Java string to a Pascal string or character.
 * @param targetType the target type specification.
 * @param javaValue the Java string.
 * @return the Pascal string or character.
 */
protected Object toPascal(TypeSpec targetType, Object javaValue)
{
    if (javaValue instanceof String) {
        String string = (String) javaValue;

        if (targetType == Predefined.charType) {
            return string.charAt(0);  // Pascal character
        }
        else if (targetType.isPascalString()) {
            Cell charCells[] = new Cell[string.length()];

            // Build an array of characters.
            for (int i = 0; i < string.length(); ++i) {
                charCells[i] = MemoryFactory.createCell(string.charAt(i));
            }

            return charCells;  // Pascal string (array of characters)
        }
        else {
            return javaValue;
        }
    }
    else {
        return javaValue;
    }
}

/**
 * Convert a Pascal string to a Java string.
 * @param targetType the target type specification
 * @param pascalValue the Pascal string.
 * @return the Java string.
 */
protected Object toJava(TypeSpec targetType, Object pascalValue)
{
    if ( (pascalValue instanceof Cell[]) &&
         (((Cell[]) pascalValue)[0].getValue() instanceof Character) )
    {
        Cell charCells[] = (Cell[]) pascalValue;
        StringBuilder string = new StringBuilder(charCells.length);

        // Build a Java string.
        for (Cell ref : charCells) {
            string.append((Character) ref.getValue());
        }

        return string.toString();  // Java string
    }
    else {
        return pascalValue;
    }
}

/**
 * Return a copy of a Pascal value.
 * @param value the value.
 * @param node the statement node.
 * @return the copy.
 */
protected Object copyOf(Object value, ICodeNode node)
{
    Object copy = null;

    if (value instanceof Integer) {
        copy = new Integer((Integer) value);
    }
    else if (value instanceof Float) {
        copy = new Float((Float) value);
```

```
        }
        else if (value instanceof Character) {
            copy = new Character((Character) value);
        }
        else if (value instanceof Boolean) {
            copy = new Boolean((Boolean) value);
        }
        else if (value instanceof String) {
            copy = new String((String) value);
        }
        else if (value instanceof HashMap) {
            copy = copyRecord((HashMap<String, Object>) value, node);
        }
        else {
            copy = copyArray((Cell[]) value, node);
        }

        return copy;
    }

    /**
     * Return a copy of a Pascal record.
     * @param value the record value hashmap.
     * @param node the statement node.
     * @return the copy of the hashmap.
     */
    private Object copyRecord(HashMap<String, Object> value, ICodeNode node)
    {
        HashMap<String, Object> copy = new HashMap<String,Object>();

        if (value != null) {
            Set<Map.Entry<String, Object>> entries = value.entrySet();
            Iterator<Map.Entry<String, Object>> it = entries.iterator();

            while (it.hasNext()) {
                Map.Entry<String, Object> entry = it.next();
                String newKey = new String(entry.getKey());
                Cell valueCell = (Cell) entry.getValue();
                Object newValue = copyOf(valueCell.getValue(), node);

                copy.put(newKey, MemoryFactory.createCell(newValue));
            }
        }
        else {
            errorHandler.flag(node, UNINITIALIZED_VALUE, this);
        }

        return copy;
    }

    /**
     * Return a copy of a Pascal array.
     * @param valueCells the array cells.
     * @param node the statement node.
     * @return the copy of the array cells.
     */
    private Cell[] copyArray(Cell valueCells[], ICodeNode node)
    {
        int length;
        Cell copy[];

        if (valueCells != null) {
            length = valueCells.length;
            copy = new Cell[length];

            for (int i = 0; i < length; ++i) {
                Cell valueCell = (Cell) valueCells[i];
                Object newValue = copyOf(valueCell.getValue(), node);
                copy[i] = MemoryFactory.createCell(newValue);
            }
        }
        else {
            errorHandler.flag(node, UNINITIALIZED_VALUE, this);
            copy = new Cell[1];
        }

        return copy;
    }

    /**
     * Runtime range check.
     * @param node the root node of the expression subtree to check.
     * @param type the target type specification.
     * @param value the value.
     * @return the value to use.
     */
    protected Object checkRange(ICodeNode node, TypeSpec type, Object value)
    {
        if (type.getForm() == SUBRANGE) {
            int minValue = (Integer) type.getAttribute(SUBRANGE_MIN_VALUE);
            int maxValue = (Integer) type.getAttribute(SUBRANGE_MAX_VALUE);

            if (((Integer) value) < minValue) {
                errorHandler.flag(node, VALUE_RANGE, this);
                return minValue;
            }
            else if (((Integer) value) > maxValue) {
                errorHandler.flag(node, VALUE_RANGE, this);
                return maxValue;
            }
            else {
                return value;
            }
        }
        else {
            return value;
        }
    }
}
```

Method `toPascal()` converts a Java string value to a Pascal string or character value. A Pascal character value is a `Cell` object that contains the value. A Pascal string is an array of `Cell` objects, one for each character of the string. Method `toJava()` does the reverse: it converts a Pascal string to a Java string.

Method `copyOf()` creates and returns a copy of a Pascal value. For a scalar or string value, it returns the corresponding Java object. Otherwise, it calls `copyArray()` or `copyRecord()` for a Pascal array or record value, respectively. Method `copyArray()` makes a copy of each `Cell` object in the array, whereas method `copyRecord()` makes a copy of the record's memory map. These copy methods check for uninitialized values.

Method `checkRange()` does runtime range checking. If verifies that a value of a subrange type is within the minimum and maximum values of the subrange. If not, the method calls the runtime error handler.

[Listing 12-14](#) shows the methods of class `StatementExecutor` that the statement executor subclasses will use to send runtime messages for debugging and tracing the source program. Since Chapter 6, the class included method `sendSourceLineMessage()`. Now add several more similar methods.

Listing 12-14: Methods of class `StatementExecutor` for sending runtime messages

```java
/**
 * Send a message about an assignment operation.
 * @param node the parse tree node.
 * @param variableName the name of the target variable.
 * @param value the value of the expression.
 */
protected void sendAssignMessage(ICodeNode node, String variableName,
                                 Object value)
{
    Object lineNumber = getLineNumber(node);

    // Send an ASSIGN message.
    if (lineNumber != null) {
        sendMessage(new Message(ASSIGN, new Object[] {lineNumber,
                                                      variableName,
                                                      value}));
    }
}

/**
 * Send a message about a value fetch operation.
 * @param node the parse tree node.
 * @param variableName the name of the variable.
 * @param value the value of the expression.
 */
protected void sendFetchMessage(ICodeNode node, String variableName,
                                Object value)
{
    Object lineNumber = getLineNumber(node);

    // Send a FETCH message.
    if (lineNumber != null) {
        sendMessage(new Message(FETCH, new Object[] {lineNumber,
                                                     variableName,
                                                     value}));
    }
}

/**
 * Send a message about a call to a declared procedure or function.
 * @param node the parse tree node.
 * @param variableName the name of the variable.
 * @param value the value of the expression.
 */
protected void sendCallMessage(ICodeNode node, String routineName)
{
    Object lineNumber = getLineNumber(node);

    // Send a CALL message.
    if (lineNumber != null) {
        sendMessage(new Message(CALL, new Object[] {lineNumber,
                                                    routineName}));
    }
}

/**
 * Send a message about a return from a declared procedure or function.
 * @param node the parse tree node.
 * @param variableName the name of the variable.
 * @param value the value of the expression.
 */
protected void sendReturnMessage(ICodeNode node, String routineName)
{
    Object lineNumber = getLineNumber(node);

    // Send a RETURN message.
    if (lineNumber != null) {
        sendMessage(new Message(RETURN, new Object[] {lineNumber,
                                                      routineName}));
    }
}

/**
 * Get the source line number of a parse tree node.
 * @param node the parse tree node.
 * @return the line number.
 */
private Object getLineNumber(ICodeNode node)
{
    Object lineNumber = null;

    // Go up the parent links to look for a line number.
    while ((node != null) &&
           ((lineNumber = node.getAttribute(LINE)) == null)) {
        node = node.getParent();
    }

    return lineNumber;
}
```

Method `sendAssignMessage()` sends a message when an assignment statement is executed. Method `sendFetchMessage()` sends a message when a value is fetched from a memory map. Together, these methods help trace a variable's value at run time. Methods `sendCallMessage()` and `sendReturnMessage()` send a message when a procedure or function is called and returns, respectively. As you'll see later, command-line flags determine whether or not the messages these methods send are received and printed.

Method `getLineNumber()` examines a statement parse tree node and, if necessary, the node's ancestors, to return the source line number of the statement.

The Assignment and Expression Executors

[Listing 12-15](#) shows the updated `execute()` method of class `AssignmentExecutor`.

```java
/**
 * Execute an assignment statement.
 * @param node the root node of the statement.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    // The ASSIGN node&apos;s children are the target variable
    // and the expression.
    ArrayList<ICodeNode> children = node.getChildren();
    ICodeNode variableNode = children.get(0);
    ICodeNode expressionNode = children.get(1);
    SymTabEntry variableId = (SymTabEntry) variableNode.getAttribute(ID);

    // Execute the target variable to get its reference and
    // execute the expression to get its value.
    ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
    Cell targetCell =
            (Cell) expressionExecutor.executeVariable(variableNode);
    TypeSpec targetType = variableNode.getTypeSpec();
    TypeSpec valueType  = expressionNode.getTypeSpec().baseType();
    Object value = expressionExecutor.execute(expressionNode);

    assignValue(node, variableId, targetCell, targetType, value, valueType);
    ++executionCount;

    return null;
}

/**
 * Assign a value to a target cell.
 * @param node the ancester parse tree node of the assignment.
 * @param targetId the symbol table entry of the target variable or parm.
 * @param targetCell the target cell.
 * @param targetType the target type.
 * @param value the value to assign.
 * @param valueType the value type.
 */
protected void assignValue(ICodeNode node, SymTabEntry targetId,
                           Cell targetCell, TypeSpec targetType,
                           Object value, TypeSpec valueType)
{
    // Range check.
    value = checkRange(node, targetType, value);

    // Set the target&apos;s value.
    // Convert an integer value to real if necessary.
    if ((targetType == Predefined.realType) &&
        (valueType  == Predefined.integerType))
    {
        targetCell.setValue(new Float(((Integer) value).intValue()));
    }

    // String assignment:
    //   target length < value length: truncate the value
    //   target length > value length: blank pad the value
    else if (targetType.isPascalString()) {
        int targetLength =
                (Integer) targetType.getAttribute(ARRAY_ELEMENT_COUNT);
        int valueLength =
                (Integer) valueType.getAttribute(ARRAY_ELEMENT_COUNT);
        String stringValue = (String) value;

        // Truncate the value string.
        if (targetLength < valueLength) {
            stringValue = stringValue.substring(0, targetLength);
        }

        // Pad the value string with blanks at the right end.
        else if (targetLength > valueLength) {
            StringBuilder buffer = new StringBuilder(stringValue);

            for (int i = valueLength; i < targetLength; ++i) {
                buffer.append(" ");
            }

            stringValue = buffer.toString();
        }

        targetCell.setValue(copyOf(toPascal(targetType, stringValue),
                                   node));
    }

    // Simple assignment.
    else {
        targetCell.setValue(copyOf(toPascal(targetType, value), node));
    }

    sendAssignMessage(node, targetId.getName(), value);
}
```

The `execute()` method calls `expressionExecutor.executeVariable()` to obtain the target variable's memory cell. It calls method `assignValue()` to do the actual assignment.

Method `assignValue()` calls `checkRange()` to perform range checking. It always makes a copy of the expression value to assign to the target memory cell. If necessary, it converts an integer value to a real value. If the expression value is a string, the method can truncate the string (the target string type is shorter) or pad the value with blanks (the target string type is longer).

Update class `ExpressionExecutor` to accommodate runtime memory management. In its `execute()` method, the `switch` statement processes the various expression parse tree nodes. Listing 12-16 shows updates to the VARIABLE and INTEGER_CONSTANT cases and the new CALL case.

```java
    switch (nodeType) {

        case VARIABLE: {

            // Return the variable&apos;s value.
            return executeValue(node);
        }

        case INTEGER_CONSTANT: {
```

```
            TypeSpec type = node.getTypeSpec();
            Integer value = (Integer) node.getAttribute(VALUE);

            // If boolean, return true or false.
            // Else return the integer value.
            return type == Predefined.booleanType
                        ? value == 1  // true or false
                        : value;      // integer value
        }

        ...

        case CALL: {

            // Execute a function call.
            SymTabEntry functionId = (SymTabEntry) node.getAttribute(ID);
            RoutineCode routineCode =
                (RoutineCode) functionId.getAttribute(ROUTINE_CODE);
            CallExecutor callExecutor = new CallExecutor(this);
            Object value = callExecutor.execute(node);

            // If it was a declared function, obtain the function value
            // from its name.
            if (routineCode == DECLARED) {
                String functionName = functionId.getName();
                int nestingLevel = functionId.getSymTab().getNestingLevel();
                ActivationRecord ar = runtimeStack.getTopmost(nestingLevel);
                Cell functionValueCell = ar.getCell(functionName);
                value = functionValueCell.getValue();

                sendFetchMessage(node, functionId.getName(), value);
            }

            // Return the function value.
            return value;
        }

        ...

    }
```

In the `execute()` method, the `VARIABLE` case calls method `executeValue()`. The `CALL` case executes a function call by calling `callExecutor.execute()`. If it was a declared function, the method extracts the function value from its memory cell by first obtaining the function name's nesting level with the calls `functionId.getSymTab().getNestingLevel()`. It then uses the nesting level and the function's name in calls to `runtimeStack.getTopmost()`, `ar.getCell()`, and `cell.getValue()` to access the appropriate activation record, memory cell, and value. A standard function call returns its value directly.

Listing 12-17 shows method `executeValue()` which returns a variable's value.

Listing 12-17: **Method `executeValue()` of class `ExpressionExecutor`**

```
    /**
     * Return a variable&apos;s value.
     * @param node ICodeNode
     * @return Object
     */
    private Object executeValue(ICodeNode node)
    {
        SymTabEntry variableId = (SymTabEntry) node.getAttribute(ID);
        String variableName = variableId.getName();
        TypeSpec variableType = variableId.getTypeSpec();

        // Get the variable&apos;s value.
        Cell variableCell = executeVariable(node);
        Object value = variableCell.getValue();

        if (value != null) {
            value = toJava(variableType, value);
        }

        // Uninitialized value error: Use a default value.
        else {
            errorHandler.flag(node, UNINITIALIZED_VALUE, this);

            value = BackendFactory.defaultValue(variableType);
            variableCell.setValue(value);
        }

        sendFetchMessage(node, variableName, value);
        return value;
    }
```

Method `executeValue()` calls `executeVariable()` to obtain the variable's cell reference. It calls `toJava()` to convert a Pascal value to a Java value for internal use. If the value is uninitialized, the method the new `BackendFactory.defaultValue()` method which returns the data type's default value. The method calls `sendFetchMessage()` before returning the variable's value.

Listing 12-18 shows method `executeVariable()` which returns a reference to a variable's cell.

Listing 12-18: **Method `executeVariable()` of class `ExpressionExecutor`**

```
    /**
     * Execute a variable and return the reference to its cell.
     * @param node the variable node.
     * @return the reference to the variable&apos;s cell.
     */
    public Cell executeVariable(ICodeNode node)
    {
        SymTabEntry variableId = (SymTabEntry) node.getAttribute(ID);
        String variableName = variableId.getName();
        TypeSpec variableType = variableId.getTypeSpec();
        int nestingLevel = variableId.getSymTab().getNestingLevel();

        // Get the variable reference from the appropriate activation record.
        ActivationRecord ar = runtimeStack.getTopmost(nestingLevel);
        Cell variableCell = ar.getCell(variableName);

        ArrayList<ICodeNode> modifiers = node.getChildren();

        // Reference to a reference: Use the original reference.
        if (variableCell.getValue() instanceof Cell) {
            variableCell = (Cell) variableCell.getValue();
        }
```

```
            // Execute any array subscripts or record fields.
            for (ICodeNode modifier : modifiers) {
                ICodeNodeType nodeType = modifier.getType();

                // Subscripts.
                if (nodeType == SUBSCRIPTS) {
                    ArrayList<ICodeNode> subscripts = modifier.getChildren();

                    // Compute a new reference for each subscript.
                    for (ICodeNode subscript : subscripts) {
                        TypeSpec indexType =
                            (TypeSpec) variableType.getAttribute(ARRAY_INDEX_TYPE);
                        int minIndex = indexType.getForm() == SUBRANGE
                            ? (Integer) indexType.getAttribute(SUBRANGE_MIN_VALUE)
                            : 0;

                        int value = (Integer) execute(subscript);
                        value = (Integer) checkRange(node, indexType, value);

                        int index = value - minIndex;
                        variableCell = ((Cell[]) variableCell.getValue())[index];
                        variableType = (TypeSpec)
                            variableType.getAttribute(ARRAY_ELEMENT_TYPE);
                    }
                }

                // Field.
                else if (nodeType == FIELD) {
                    SymTabEntry fieldId = (SymTabEntry) modifier.getAttribute(ID);
                    String fieldName = fieldId.getName();

                    // Compute a new reference for the field.
                    HashMap<String, Cell> map =
                        (HashMap<String, Cell>) variableCell.getValue();
                    variableCell = map.get(fieldName);
                    variableType = fieldId.getTypeSpec();
                }
            }

            return variableCell;
        }
}
```

Method `executeVariable()` first obtains the nesting level of the variable's name with a call to `id.getSymTab().getNestingLevel()`. It uses the nesting level and the variable's name in calls to `runtimeStack.getTopmost()` and `ar.getCell(name)` to access the appropriate activation record and a reference to the variable's memory cell. If it turns out that the value the reference points to is itself a reference (i.e., `variableCell.getValue() instanceof Cell` is true), then the statement

```
            variableCell = (Cell) variableCell.getValue();
```

strips off the outer reference in order to use the original reference.

Starting with the `variableCell` reference, method `executeVariable()` loops to execute any subscripts or fields. For each subscript, the method computes a new `Cell` reference by indexing into the array of `Cell` objects. It calls `checkRange()` to perform a range check of the subscript value. For each field, the method obtains a reference to the field's memory cell from record value's memory map. After exiting the loop, the method returns a reference to the variable's memory cell.

Listing 12-19 shows the new static method `BackendFactory.defaultValue()`, which returns a default value for a given data type.

Listing 12-19: Method `defaultValue()` of class `BackendFactory`

```
/**
 * Return the default value for a data type.
 * @param type the data type.
 * @return the type descriptor.
 */
public static Object defaultValue(TypeSpec type)
{
    type = type.baseType();

    if (type == Predefined.integerType) {
        return new Integer(0);
    }
    else if (type == Predefined.realType) {
        return new Float(0.0f);
    }
    else if (type == Predefined.booleanType) {
        return new Boolean(false);
    }
    else if (type == Predefined.charType) {
        return new Character('#');
    }
    else /* string */ {
        return new String("#");
    }
}
```

Listing 12-20 shows updates to method `executeBinaryOperator()` in class `ExpressionExecutor` to handle character and string values.

Listing 12-20: Updates to method `executeVariable()` of class `ExpressionExecutor`

```
private Object executeBinaryOperator(ICodeNode node,
                                     ICodeNodeTypeImpl nodeType)
{
    ...

    boolean integerMode = false;
    boolean characterMode = false;
    boolean stringMode = false;

    if ((operand1 instanceof Integer) && (operand2 instanceof Integer)) {
        integerMode = true;
    }
    else if ( ( (operand1 instanceof Character) ||
                ( (operand1 instanceof String) &&
                  (((String) operand1).length() == 1) )
              ) &&
              ( (operand2 instanceof Character) ||
```

```
                 ( (operand2 instanceof String) &&
                   (((String) operand2).length() == 1) )
             )
         ) {
        characterMode = true;
    }
    else if ((operand1 instanceof String) && (operand2 instanceof String)) {
        stringMode = true;
    }

    ...

    // ====================
    // Relational operators
    // ====================

    ...

    else if (characterMode) {
        int value1 = operand1 instanceof Character
                        ? (Character) operand1
                        : ((String) operand1).charAt(0);
        int value2 = operand2 instanceof Character
                        ? (Character) operand2
                        : ((String) operand2).charAt(0);

        // Character operands.
        switch (nodeType) {
            case EQ: return value1 == value2;
            case NE: return value1 != value2;
            case LT: return value1 <  value2;
            case LE: return value1 <= value2;
            case GT: return value1 >  value2;
            case GE: return value1 >= value2;
        }
    }
    else if (stringMode) {
        String value1 = (String) operand1;
        String value2 = (String) operand2;

        // String operands.
        int comp = value1.compareTo(value2);
        switch (nodeType) {
            case EQ: return comp == 0;
            case NE: return comp != 0;
            case LT: return comp <  0;
            case LE: return comp <= 0;
          case GT: return comp >  0;
          case GE: return comp >= 0;
        }
    }
    ...
}
```

Method `ExpressionExecutor` sets the values of boolean variables `integerMode`, `characterMode`, and `stringMode` according to the types of the operands. It executes the relational operators for character and string operands.

Executing Procedure and Function Calls

Executing procedure and function calls involve parameter passing, calling a routine, and returning from a routine.

Parameter Passing

As you've already seen, Pascal parameters can be passed by value (the default) or by reference (`VAR` parameters).

Figure 12-6 shows parameter passing.

Figure 12-6 shows the activation records for `main` and procedure `proc` on the runtime stack. When actual parameter values are passed by value, the executor copies each value into the corresponding formal parameter. When the value is an array or a record, this can be a time-consuming operation. However, when actual parameter values are passed by reference, the executor simply sets each corresponding formal `VAR` parameter to point to the actual parameter value, and there is no copying of the value.

Calling Procedures and Functions

Figure 12-7 shows the statement executor subclasses for calling declared and standard procedures and functions.

Class `CallDeclaredExecutor` has private method `executeActualParms()` that evaluates and passes actual parameters. Class `CallStandardExecutor` has private methods that execute calls to the standard routines. Both are subclasses of class `CallExecutor`, which is itself a subclass of class `StatementExecutor` (see Figure 6-1).

Listing 12-21 shows the `execute()` method of class `CallExecutor`.

Figure 12-6: Parameter passing for the call `main` $\Rightarrow$ `proc`
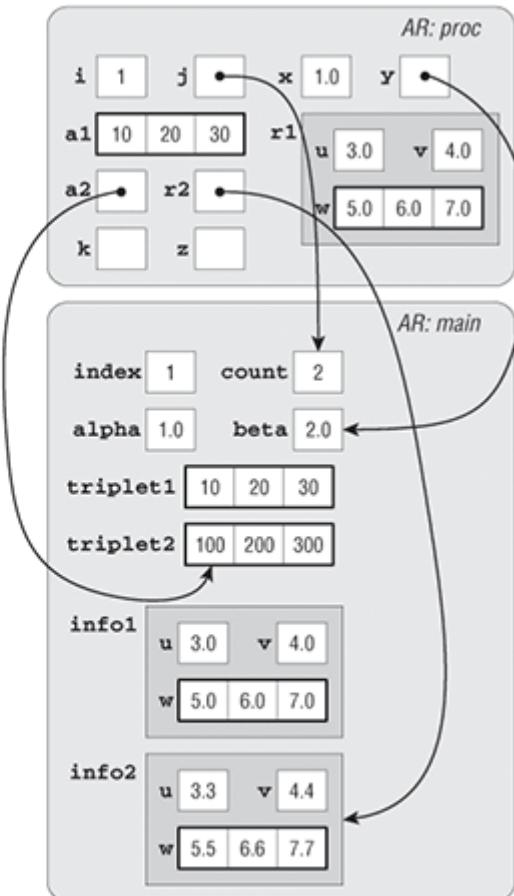
```pascal
PROGRAM main;

TYPE
  arr = ARRAY[1..3] OF integer;
  rec = RECORD
          u, v : real;
          w : arr
        END;
VAR
  index, count : integer;
  alpha, beta : real;
  triplet1, triplet2 : arr;
  info1, info2 : rec;
...
PROCEDURE proc(i : integer;
               x : real;
               VAR j : integer;
               VAR y : real;
               a1 : arr;
               r1 : rec;
               VAR a2 : arr,
               VAR r2 : rec);

  VAR
    k : integer;
    z : real;

  BEGIN {proc}
    ...
  END;

BEGIN {main}
  ...
  proc(index, alpha, count, beta,
       triplet1, info1,
       triplet2, info2);
  ...
END.
```

AR: proc

| i | 1 | j | • | x | 1.0 | y | • |

a1	10	20	30
r1	u	3.0	v	4.0
	w	5.0	6.0	7.0

| a2 | • | r2 | • |

| k | | z | |

AR: main

| index | 1 | count | 2 |

| alpha | 1.0 | beta | 2.0 |

| triplet1 | 10 | 20 | 30 |

| triplet2 | 100 | 200 | 300 |

info1
| u | 3.0 | v | 4.0 |
| w | 5.0 | 6.0 | 7.0 |

info2
| u | 3.3 | v | 4.4 |
| w | 5.5 | 6.6 | 7.7 |

Listing 12-21: Method `execute()` of class `CallExecutor`
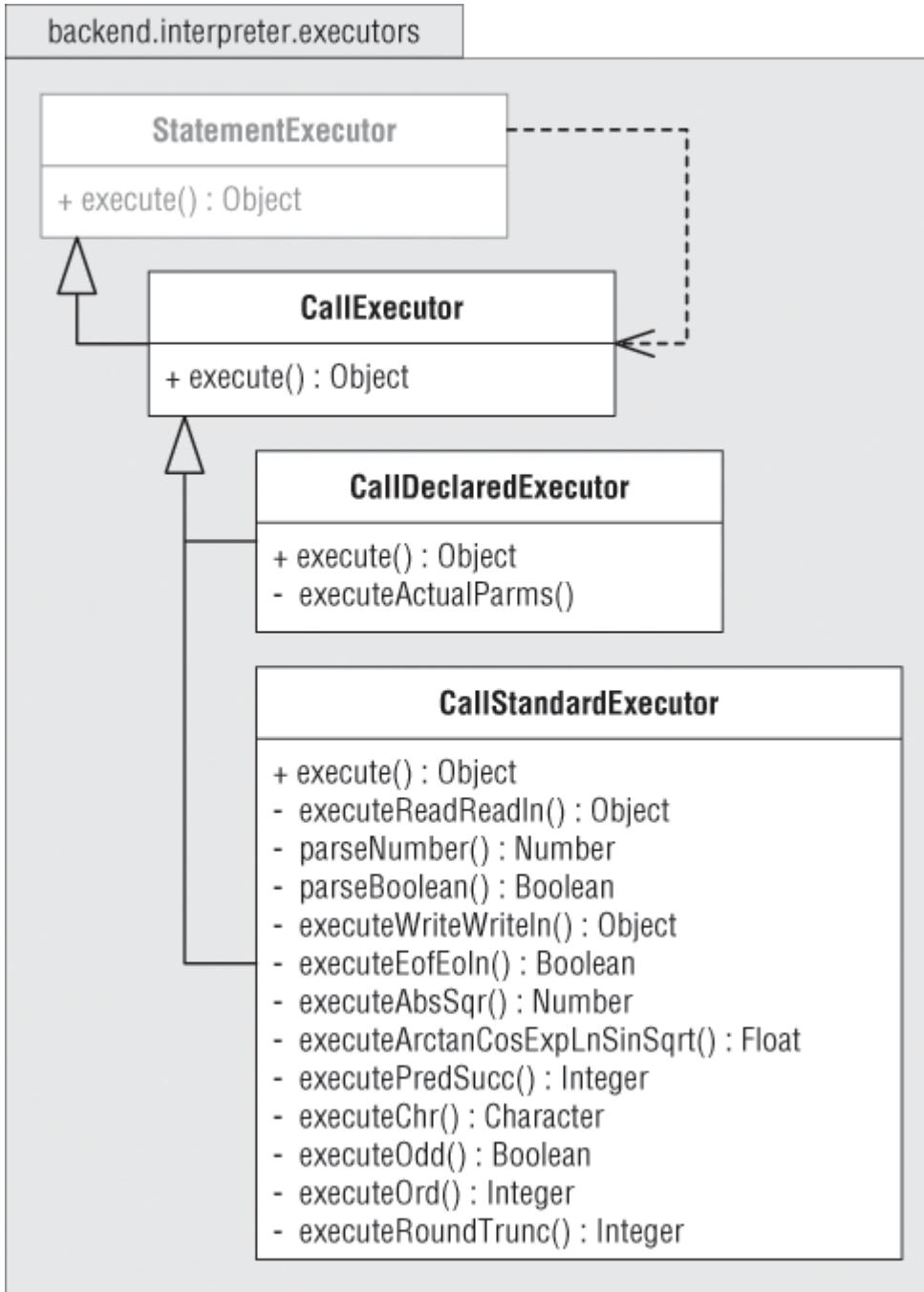
```java
/**
 * Execute procedure or function call statement.
 * @param node the root node of the call.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    SymTabEntry routineId = (SymTabEntry) node.getAttribute(ID);
    RoutineCode routineCode =
                (RoutineCode) routineId.getAttribute(ROUTINE_CODE);
    CallExecutor callExecutor = routineCode == DECLARED
                    ? new CallDeclaredExecutor(this)
                    : new CallStandardExecutor(this);

    ++executionCount;  // count the call statement
    return callExecutor.execute(node);
}
```

Figure 12-7: Classes for calling declared and standard procedures and functions

backend.interpreter.executors

StatementExecutor

+ execute() : Object

CallExecutor

+ execute() : Object

CallDeclaredExecutor

+ execute() : Object
- executeActualParms()

CallStandardExecutor

+ execute() : Object
- executeReadReadln() : Object
- parseNumber() : Number
- parseBoolean() : Boolean
- executeWriteWriteln() : Object
- executeEofEoln() : Boolean
- executeAbsSqr() : Number
- executeArctanCosExpLnSinSqrt() : Float
- executePredSucc() : Integer
- executeChr() : Character
- executeOdd() : Boolean
- executeOrd() : Integer
- executeRoundTrunc() : Integer

The `execute()` method checks the symbol table entry of the procedure or function name to determine whether the routine is declared or standard. It then calls the `execute()` method of either class `CallDeclaredExecutor` or class `CallStandardExecutor`.

Listing 12-22 shows the `execute()` and `executeActualParms()` methods of class `CallDeclaredExecutor`.

Listing 12-22: Methods `execute()` and `executeActualParms()` of class `CallDeclaredExecutor`

```
/**
 * Execute a call to a declared procedure or function.
 * @param node the CALL node.
 * @return null.
 */
public Object execute(ICodeNode node)
{
    SymTabEntry routineId = (SymTabEntry) node.getAttribute(ID);
    ActivationRecord newAr =
        MemoryFactory.createActivationRecord(routineId);

    // Execute any actual parameters and initialize
    // the formal parameters in the new activation record.
    if (node.getChildren().size() > 0) {
        ICodeNode parmsNode = node.getChildren().get(0);
        ArrayList<ICodeNode> actualNodes = parmsNode.getChildren();
        ArrayList<SymTabEntry> formalIds =
            (ArrayList<SymTabEntry>) routineId.getAttribute(ROUTINE_PARMS);
        executeActualParms(actualNodes, formalIds, newAr);
    }

    // Push the new activation record.
    runtimeStack.push(newAr);

    sendCallMessage(node, routineId.getName());

    // Get the root node of the routine's intermediate code.
    ICode iCode = (ICode) routineId.getAttribute(ROUTINE_ICODE);
    ICodeNode rootNode = iCode.getRoot();

    // Execute the routine.
    StatementExecutor statementExecutor = new StatementExecutor(this);
    Object value = statementExecutor.execute(rootNode);
```

```
            // Pop off the activation record.
            runtimeStack.pop();

            sendReturnMessage(node, routineId.getName());
            return value;
        }

    /**
     * Execute the actual parameters of a call.
     * @param actualNodes the list of nodes of the actual parms.
     * @param formalIds the list of symbol table entries of the formal parms.
     * @param newAr the new activation record.
     */
    private void executeActualParms(ArrayList<ICodeNode> actualNodes,
                                    ArrayList<SymTabEntry> formalIds,
                                    ActivationRecord newAr)
    {
        ExpressionExecutor expressionExecutor = new ExpressionExecutor(this);
        AssignmentExecutor assignmentExecutor = new AssignmentExecutor(this);

        for (int i = 0; i < formalIds.size(); ++i) {
            SymTabEntry formalId = formalIds.get(i);
            Definition formalDefn = formalId.getDefinition();
            Cell formalCell= newAr.getCell(formalId.getName());
            ICodeNode actualNode = actualNodes.get(i);

            // Value parameter.
            if (formalDefn == VALUE_PARM) {
                TypeSpec formalType = formalId.getTypeSpec();
                TypeSpec valueType  = actualNode.getTypeSpec().baseType();
                Object value = expressionExecutor.execute(actualNode);

                assignmentExecutor.assignValue(actualNode, formalId,
                                               formalCell, formalType,
                                               value, valueType);
            }

            // VAR parameter.
            else {
                Cell actualCell=
                    (Cell) expressionExecutor.executeVariable(actualNode);
                formalCell.setValue(actualCell);
            }
        }
    }
}
```

Method `execute()` creates a new activation record for the routine that it's calling. It calls `executeActualParms()` to execute any actual parameters and set the memory map of the new activation record. The method calls `runtimeStack.push()` to push the activation record onto the runtime stack, obtains the root node of the routine's parse tree from the routine name's symbol table entry, and calls `statementExecutor.execute()` to execute the routine. Upon returning form the routine, the method calls `runtimeStack.pop()` to pop off the routine's activation record. The method calls `sendCallMessage()` and `sendReturnMessage()` to send call and return trace messages.

Method `executeActualParms()` loops over the formal parameter list to process each formal parameter and the corresponding actual parameter.

If the actual parameter is being passed by value, the method calls `expressionExecutor.execute()` and `assignmentExecutor.assignValue()` to assign the actual parameter value to the formal parameter.

If the actual parameter is being passed by reference, method `executeActualParms()` calls `expressionExecutor.executeVariable()` to obtain a reference to the actual parameter's memory cell, and then it calls `formalCell.setValue()` to set that reference into the formal parameter's memory cell inside the called routine's new activation record.

Listing 12-23 shows the `execute()` method of class `CallStandardExecutor`.

Listing 12-23: Method `execute()` of class `CallStandardExecutor`

```
    /**
     * Execute a call to a standard procedure or function.
     * @param node the CALL node.
     * @return the function value, or null for a procedure call.
     */
    public Object execute(ICodeNode node)
    {
        SymTabEntry routineId = (SymTabEntry) node.getAttribute(ID);
        RoutineCode routineCode =
                    (RoutineCode) routineId.getAttribute(ROUTINE_CODE);
        TypeSpec type = node.getTypeSpec();
        expressionExecutor = new ExpressionExecutor(this);
        ICodeNode actualNode = null;

        // Get the actual parameters of the call.
        if (node.getChildren().size() > 0) {
            ICodeNode parmsNode = node.getChildren().get(0);
            actualNode = parmsNode.getChildren().get(0);
        }

        switch ((RoutineCodeImpl) routineCode) {
            case READ:
            case READLN:   return executeReadReadln(node, routineCode);

            case WRITE:
            case WRITELN:  return executeWriteWriteln(node, routineCode);

            case EOF:
            case EOLN:     return executeEofEoln(node, routineCode);

            case ABS:
            case SQR:      return executeAbsSqr(node, routineCode, actualNode);

            case ARCTAN:
            case COS:
            case EXP:
            case LN:
            case SIN:
            case SQRT:     return executeArctanCosExpLnSinSqrt(node, routineCode,
                                                              actualNode);

            case PRED:
            case SUCC:     return executePredSucc(node, routineCode,
                                                  actualNode, type);

            case CHR:      return executeChr(node, routineCode, actualNode);
```

```
            case ODD:     return executeOdd(node, routineCode, actualNode);
            case ORD:     return executeOrd(node, routineCode, actualNode);

            case ROUND:
            case TRUNC:   return executeRoundTrunc(node, routineCode,
                                                   actualNode);

            default:      return null;  // should never get here
        }
    }
}
```

Just as you had to write *ad hoc* methods in Chapter 11 to parse calls to the standard Pascal procedures and functions, now write *ad hoc* methods to execute the calls. Method `execute()` determines which method to call.

Listing 12-24 shows method `executeReadReadln()` and the parsing methods.

Listing 12-24: **Method `executeReadReadln()` and the parsing methods of class `CallStandardExecutor`**

```
/**
 * Execute a call to read or readln.
 * @param callNode the CALL node.
 * @param routineCode the routine code.
 * @return null.
 */
private Object executeReadReadln(ICodeNode callNode,
                                 RoutineCode routineCode)
{
    ICodeNode parmsNode = callNode.getChildren().size() > 0
                            ? callNode.getChildren().get(0)
                            : null;

    if (parmsNode != null) {
        ArrayList<ICodeNode> actuals = parmsNode.getChildren();

        // Loop to process each actual parameter.
        for (ICodeNode actualNode : actuals) {
            TypeSpec type = actualNode.getTypeSpec();
            TypeSpec baseType = type.baseType();
            Cell variableCell =
                    (Cell) expressionExecutor.executeVariable(actualNode);
            Object value;

            // Read a value of the appropriate type from the standard input.
            try {
                if (baseType == Predefined.integerType) {
                    Token token = standardIn.nextToken();
                    value = (Integer) parseNumber(token, baseType);
                }
                else if (baseType == Predefined.realType) {
                    Token token = standardIn.nextToken();
                    value = (Float) parseNumber(token, baseType);
                }
                else if (baseType == Predefined.booleanType) {
                    Token token = standardIn.nextToken();
                    value = parseBoolean(token);
                }
                else if (baseType == Predefined.charType) {
                    char ch = standardIn.nextChar();
                    if ((ch == Source.EOL) || (ch == Source.EOF)) {
                        ch = ' ';
                    }
                    value = ch;
                }
                else {
                    throw new Exception();
                }
            }
            catch (Exception ex) {
                errorHandler.flag(callNode, INVALID_INPUT,
                                  CallStandardExecutor.this);

                if (type == Predefined.realType) {
                    value = 0.0f;
                }
                else if (type == Predefined.charType) {
                    value = ' ';
                }
                else if (type == Predefined.booleanType) {
                    value = false;
                }
                else {
                    value = 0;
                }
            }

            // Range check and set the value.
            value = checkRange(callNode, type, value);
            variableCell.setValue(value);

            SymTabEntry actualId =
                        (SymTabEntry) actualNode.getAttribute(ID);
            sendAssignMessage(callNode, actualId.getName(), value);
        }
    }

    // Skip the rest of the input line for readln.
    if (routineCode == READLN) {
        try {
            standardIn.skipToNextLine();
        }
        catch (Exception ex) {
            errorHandler.flag(callNode, INVALID_INPUT,
                              CallStandardExecutor.this);
        }
    }

    return null;
}

/**
 * Parse an integer or real value from the standard input.
 * @param token the current input token.
 * @param type the input value type.
 * @return the integer or real value.
 * @throws Exception if an error occurred.
 */
private Number parseNumber(Token token, TypeSpec type)
    throws Exception
{
```

```
            TokenType tokenType = token.getType();
            TokenType sign = null;

            // Leading sign?
            if ((tokenType == PLUS) || (tokenType == MINUS)) {
                sign = tokenType;
                token = standardIn.nextToken();
                tokenType = token.getType();
            }

            // Integer value.
            if (tokenType == INTEGER) {
                Number value = sign == MINUS ? -((Integer) token.getValue())
                                             : (Integer) token.getValue();
                return type == Predefined.integerType
                           ? value
                           : new Float(((Integer) value).intValue());
            }

            // Real value.
            else if (tokenType == REAL) {
                Number value = sign == MINUS ? -((Float) token.getValue())
                                             : (Float) token.getValue();
                return type == Predefined.realType
                           ? value
                           : new Integer(((Float) value).intValue());
            }

            // Bad input.
            else {
                throw new Exception();
            }
        }

        /**
         * Parse a boolean value from the standard input.
         * @param token the current input token.
         * @param type the input value type.
         * @return the boolean value.
         * @throws Exception if an error occurred.
         */
        private Boolean parseBoolean(Token token)
            throws Exception
        {
            if (token.getType() == IDENTIFIER) {
                String text = token.getText();

                if (text.equalsIgnoreCase("true")) {
                    return new Boolean(true);
                }
                else if (text.equalsIgnoreCase("false")) {
                    return new Boolean(false);
                }
                else {
                    throw new Exception();
                }
            }
            else {
                throw new Exception();
            }
        }
```

Method `executeReadReadln()` executes calls to the standard `read` and `readln` procedures. Because you implemented `standardIn` as an instance of `PascalScanner` (see [Listing 12-12](#)), the method calls `standardIn.nextToken()` to read the next integer, real, or boolean token from the input text, or `standardIn.nextChar()` to read the next input character. It calls `parseNumber()` or `parseBoolean()` to parse integer, real, or boolean input values. If there is a type mismatch, the method calls the runtime error handler and sets a default value depending on the type of the actual parameter.

[Listing 12-25](#) shows method `executeWriteWriteln()`.

[Listing 12-25:](#) Method `executeWriteWriteln()` of class `CallStandardExecutor`

```
        /**
         * Execute a call to write or writeln.
         * @param callNode the CALL node.
         * @param routineCode the routine code.
         * @return null.
         */
        private Object executeWriteWriteln(ICodeNode callNode,
                                           RoutineCode routineCode)
        {
            ICodeNode parmsNode = callNode.getChildren().size() > 0
                                      ? callNode.getChildren().get(0)
                                      : null;

            if (parmsNode != null) {
                ArrayList<ICodeNode> actuals = parmsNode.getChildren();

                // Loop to process each WRITE_PARM actual parameter node.
                for (ICodeNode writeParmNode : actuals) {
                    ArrayList<ICodeNode> children = writeParmNode.getChildren();
                    ICodeNode exprNode = children.get(0);
                    TypeSpec dataType = exprNode.getTypeSpec().baseType();
                    String typeCode = dataType.isPascalString()        ? "s"
                        : dataType == Predefined.integerType ? "d"
                        : dataType == Predefined.realType    ? "f"
                        : dataType == Predefined.booleanType ? "s"
                        : dataType == Predefined.charType    ? "c"
                        :                                      "s";
                    Object value = expressionExecutor.execute(exprNode);

                    if ((dataType == Predefined.charType) &&
                        (value instanceof String))
                    {
                        value = ((String) value).charAt(0);
                    }

                    // Java format string.
                    StringBuilder format = new StringBuilder("%");

                    // Process any field width and precision values.
                    if (children.size() > 1) {
                        int w = (Integer) children.get(1).getAttribute(VALUE);
                        format.append(w == 0 ? 1 : w);
                    }
                    if (children.size() > 2) {
                        int p = (Integer) children.get(2).getAttribute(VALUE);
```

```
                    format.append(".");
                    format.append(p == 0 ? 1 : p);
                }

                format.append(typeCode);

                // Write the formatted value to the standard output.
                standardOut.printf(format.toString(), value);
                standardOut.flush();
            }
        }

        // Line feed for writeln.
        if (routineCode == WRITELN) {
            standardOut.println();
            standardOut.flush();
        }

        return null;
    }
```

Method `executeWriteWriteln()` executes calls to the standard `write` and `writeln` procedures. It builds a format string for each value which it then outputs with a call to `standardOut.printf()`.

shows the remaining methods of class `CallStandardExecutor`.

** The remaining methods of class `CallStandardExecutor`**

```
    /**
     * Execute a call to eof or eoln.
     * @param callNode the CALL node.
     * @param routineCode the routine code.
     * @return true or false.
     */
    private Boolean executeEofEoln(ICodeNode callNode, RoutineCode routineCode)
    {
        try {
            if (routineCode == EOF) {
                return standardIn.atEof();
            }
            else {
                return standardIn.atEol();
            }
        }
        catch (Exception ex) {
            errorHandler.flag(callNode, INVALID_INPUT,
                              CallStandardExecutor.this);
            return true;
        }
    }

    /**
     * Execute a call to abs or sqr.
     * @param callNode the CALL node.
     * @param routineCode the routine code.
     * @param actualNode the actual parameter node.
     * @return the function value.
     */
    private Number executeAbsSqr(ICodeNode callNode, RoutineCode routineCode,
                                 ICodeNode actualNode)
    {
        Object argValue = expressionExecutor.execute(actualNode);

        if (argValue instanceof Integer) {
            int value = (Integer) argValue;
            return routineCode == ABS ? Math.abs(value) : value*value;
        }
        else {
            float value = (Float) argValue;
            return routineCode == ABS ? Math.abs(value) : value*value;
        }
    }

    /**
     * Execute a call to arctan, cos, exp, ln, sin, or sqrt.
     * @param callNode the CALL node.
     * @param routineCode the routine code.
     * @param actualNode the actual parameter node.
     * @return the function value.
     */
    private Float executeArctanCosExpLnSinSqrt(ICodeNode callNode,
                                               RoutineCode routineCode,
                                               ICodeNode actualNode)
    {
        Object argValue = expressionExecutor.execute(actualNode);
        Float value = argValue instanceof Integer ? (Integer) argValue
                                                   : (Float) argValue;

        switch ((RoutineCodeImpl) routineCode) {
            case ARCTAN: return (float) Math.atan(value);
            case COS:    return (float) Math.cos(value);
            case EXP:    return (float) Math.exp(value);
            case SIN:    return (float) Math.sin(value);

            case LN: {
                if (value > 0.0f) {
                    return (float) Math.log(value);
                }
                else {
                    errorHandler.flag(callNode,
                                      INVALID_STANDARD_FUNCTION_ARGUMENT,
                                      CallStandardExecutor.this);
                    return 0.0f;
                }
            }

            case SQRT: {
                if (value >= 0.0f) {
                    return (float) Math.sqrt(value);
                }
                else {
                    errorHandler.flag(callNode,
                                      INVALID_STANDARD_FUNCTION_ARGUMENT,
                                      CallStandardExecutor.this);
                    return 0.0f;
                }
            }

            default: return 0.0f;  // should never get here
```

```
        }
    }

    /**
     * Execute a call to pred or succ.
     * @param callNode the CALL node.
     * @param routineCode the routine code.
     * @param actualNode the actual parameter node.
     * @param type the value type.
     * @return the function value.
     */
    private Integer executePredSucc(ICodeNode callNode, RoutineCode routineCode,
                                    ICodeNode actualNode, TypeSpec type)
    {
        int value = (Integer) expressionExecutor.execute(actualNode);
        int newValue = routineCode == PRED ? --value : ++value;

        newValue = (Integer) checkRange(callNode, type, newValue);
        return newValue;
    }

    /**
     * Execute a call to chr.
     * @param callNode the CALL node.
     * @param routineCode the routine code.
     * @param actualNode the actual parameter node.
     * @return the function value.
     */
    private Character executeChr(ICodeNode callNode, RoutineCode routineCode,
                                 ICodeNode actualNode)
    {
        int value = (Integer) expressionExecutor.execute(actualNode);
        char ch = (char) value;
        return ch;
    }

    /**
     * Execute a call to odd.
     * @param callNode the CALL node.
     * @param routineCode the routine code.
     * @param actualNode the actual parameter node.
     * @return true or false.
     */
    private Boolean executeOdd(ICodeNode callNode, RoutineCode routineCode,
                               ICodeNode actualNode)
    {
        int value = (Integer) expressionExecutor.execute(actualNode);
        return (value & 1) == 1;
    }

    /**
     * Execute a call to ord.
     * @param callNode the CALL node.
     * @param routineCode the routine code.
     * @param actualNode the actual parameter node.
     * @return the function value.
     */
    private Integer executeOrd(ICodeNode callNode, RoutineCode routineCode,
                               ICodeNode actualNode)
    {
        Object value = expressionExecutor.execute(actualNode);

        if (value instanceof Character) {
            char ch = ((Character) value).charValue();
            return (int) ch;
        }
        else if (value instanceof String) {
            char ch = ((String) value).charAt(0);
            return (int) ch;
        }
        else {
            return (Integer) value;
        }
    }

    /**
     * Execute a call to round or trunc.
     * @param callNode the CALL node.
     * @param routineCode the routine code.
     * @param actualNode the actual parameter node.
     * @return the function value.
     */
    private Integer executeRoundTrunc(ICodeNode callNode,
                                      RoutineCode routineCode,
                                      ICodeNode actualNode)
    {
        float value = (Float) expressionExecutor.execute(actualNode);

        if (routineCode == ROUND) {
            return value >= 0.0f ? (int) (value + 0.5f)
                                 : (int) (value - 0.5f);
        }
        else {
            return (int) value;
        }
    }
}
```

Program 12-1: Pascal Interpreter

You've completed the back end executors in the back end. With the work you've done for the front end and the intermediate tier, you now have a working Pascal interpreter!

shows changes in the main `Pascal` class to handle the new command line flags that control the runtime debugging and tracing output.

Listing 20-27: Updates to the main `Pascal` class

```
private boolean intermediate;       // true to print intermediate code
private boolean xref;               // true to print cross-reference listing
private boolean lines;              // true to print source line tracing
private boolean assign;             // true to print value assignment tracing
private boolean fetch;              // true to print value fetch tracing
private boolean call;               // true to print routine call tracing
private boolean returnn;            // true to print routine return tracing
```

```
...

/**
 * Compile or interpret a Pascal source program.
 * @param operation either "compile" or "execute".
 * @param filePath the source file path.
 * @param flags the command line flags.
 */
public Pascal(String operation, String filePath, String flags)
{
    try {
        intermediate = flags.indexOf('i') > -1;
        xref         = flags.indexOf('x') > -1;
        lines        = flags.indexOf('l') > -1;
        assign       = flags.indexOf('a') > -1;
        fetch        = flags.indexOf('f') > -1;
        call         = flags.indexOf('c') > -1;
        returnn       = flags.indexOf('r') > -1;

        ...
    }

    ...

    private static final String LINE_FORMAT =
        ">>> AT LINE %03d\n";

    private static final String ASSIGN_FORMAT =
        ">>> AT LINE %03d: %s = %s\n";

    private static final String FETCH_FORMAT =
        ">>> AT LINE %03d: %s : %s\n";

    private static final String CALL_FORMAT =
        ">>> AT LINE %03d: CALL %s\n";

    private static final String RETURN_FORMAT =
        ">>> AT LINE %03d: RETURN FROM %s\n";

    /**
     * Listener for back end messages.
     */
    private class BackendMessageListener implements MessageListener
    {
        /**
         * Called by the back end whenever it produces a message.
         * @param message the message.
         */
        public void messageReceived(Message message)
        {
            MessageType type = message.getType();

            switch (type) {

                case SOURCE_LINE: {
                    if (lines) {
                        int lineNumber = (Integer) message.getBody();

                        System.out.printf(LINE_FORMAT, lineNumber);
                    }
                    break;
                }

                case ASSIGN: {
                    if (assign) {
                        Object body[] = (Object[]) message.getBody();
                        int lineNumber = (Integer) body[0];
                        String variableName = (String) body[1];
                        Object value = body[2];

                        System.out.printf(ASSIGN_FORMAT,
                                          lineNumber, variableName, value);
                    }
                    break;
                }

                case FETCH: {
                    if (fetch) {
                        Object body[] = (Object[]) message.getBody();
                        int lineNumber = (Integer) body[0];
                        String variableName = (String) body[1];
                        Object value = body[2];

                        System.out.printf(FETCH_FORMAT,
                                          lineNumber, variableName, value);
                    }
                    break;
                }

                case CALL: {
                    if (call) {
                        Object body[] = (Object[]) message.getBody();
                        int lineNumber = (Integer) body[0];
                        String routineName = (String) body[1];

                        System.out.printf(CALL_FORMAT,
                                          lineNumber, routineName);
                    }
                    break;
                }

                case RETURN: {
                    if (returnn) {
                        Object body[] = (Object[]) message.getBody();
                        int lineNumber = (Integer) body[0];
                        String routineName = (String) body[1];

                        System.out.printf(RETURN_FORMAT,
                                          lineNumber, routineName);
                    }
                    break;
                }

                ...
            }
        }
    }
}
```

The new command line flags for tracing output are -l, -a, -f, -c, and -r to specify printing the current source line number, an assignment, a data fetch, a routine call, and a routine return, respectively.

[Listing 12-28](#) shows an example of running the Pascal interpreter. The command line is similar to

```
java -cp classes Pascal execute newton.pas
```

Listing 12-28: Executing the Pascal interpreter

```
001 PROGRAM newton;
002
003 CONST
004     epsilon = 1e-6;
005
006 VAR
007     number : integer;
008
009 FUNCTION root(x : real) : real;
010     VAR
011         r : real;
012
013     BEGIN
014         r := 1;
015         REPEAT
016             r := (x/r + r)/2;
017         UNTIL abs(x/sqr(r) - 1) < epsilon;
018         root := r;
019     END;
020
021 PROCEDURE print(n : integer; root : real);
022     BEGIN
023         writeln('The square root of ', number:4, ' is ', root:10:6);
024     END;
025
026 BEGIN
027     REPEAT
028         writeln;
029         write('Enter new number (0 to quit): ');
030         read(number);
031
032         IF number = 0 THEN BEGIN
033             print(number, 0.0);
034         END
035         ELSE IF number < 0 THEN BEGIN
036             writeln('*** ERROR:  number < 0');
037         END
038         ELSE BEGIN
039             print(number, root(number));
040         END
041     UNTIL number = 0
042 END.

                  42 source lines.
                   0 syntax errors.
                0.09 seconds total parsing time.

Enter new number (0 to quit): 4
The square root of    4 is   2.000000

Enter new number (0 to quit): 0
The square root of    0 is   0.000000

                  34 statements executed.
                   0 runtime errors.
                1.95 seconds total execution time.
```

[Listing 12-29](#) shows the executing the same Pascal program but with all the runtime tracing flags turned on:

```
java -cp classes Pascal execute -lafcr newton.pas
```

Listing 12-29: Executing the Pascal interpreter with runtime tracing

```
001 PROGRAM newton;
002
003 CONST
004     epsilon = 1e-6;
005
006 VAR
007     number : integer;
008
009 FUNCTION root(x : real) : real;
010     VAR
011         r : real;
012
013     BEGIN
014         r := 1;
015         REPEAT
016             r := (x/r + r)/2;
017         UNTIL abs(x/sqr(r) - 1) < epsilon;
018         root := r;
019     END;
020
021 PROCEDURE print(n : integer; root : real);
022     BEGIN
023         writeln('The square root of ', number:4, ' is ', root:10:6);
024     END;
025
026 BEGIN
027     REPEAT
028         writeln;
029         write('Enter new number (0 to quit): ');
030         read(number);
031
032         IF number = 0 THEN BEGIN
033             print(number, 0.0);
034         END
035         ELSE IF number < 0 THEN BEGIN
036             writeln('*** ERROR:  number < 0');
037         END
038         ELSE BEGIN
039             print(number, root(number));
040         END
```

```
041     UNTIL number = 0
042 END.

              42 source lines.
               0 syntax errors.
            0.09 seconds total parsing time.
>>> AT LINE 026
>>> AT LINE 027
>>> AT LINE 028

>>> AT LINE 029
Enter new number (0 to quit): >>> AT LINE 030
4
>>> AT LINE 030: number = 4
>>> AT LINE 032
>>> AT LINE 032: number : 4
>>> AT LINE 035
>>> AT LINE 035: number : 4
>>> AT LINE 038
>>> AT LINE 039
>>> AT LINE 039: CALL print
>>> AT LINE 039: number : 4
>>> AT LINE 039: n = 4
>>> AT LINE 039: CALL root
>>> AT LINE 039: number : 4
>>> AT LINE 039: x = 4
>>> AT LINE 013
>>> AT LINE 014
>>> AT LINE 014: r = 1
>>> AT LINE 015
>>> AT LINE 016
>>> AT LINE 016: x : 4
>>> AT LINE 016: r : 1
>>> AT LINE 016: r : 1
>>> AT LINE 016: r = 2.5
>>> AT LINE 015: x : 4
>>> AT LINE 015: r : 2.5
>>> AT LINE 016
>>> AT LINE 016: x : 4
>>> AT LINE 016: r : 2.5
>>> AT LINE 016: r : 2.5
>>> AT LINE 016: r = 2.05
>>> AT LINE 015: x : 4
>>> AT LINE 015: r : 2.05
>>> AT LINE 016
>>> AT LINE 016: x : 4
>>> AT LINE 016: r : 2.05
>>> AT LINE 016: r : 2.05
>>> AT LINE 016: r = 2.0006099
>>> AT LINE 015: x : 4
>>> AT LINE 015: r : 2.0006099
>>> AT LINE 016
>>> AT LINE 016: x : 4
>>> AT LINE 016: r : 2.0006099
>>> AT LINE 016: r : 2.0006099
>>> AT LINE 016: r = 2.0
>>> AT LINE 015: x : 4
>>> AT LINE 015: r : 2.0
>>> AT LINE 018
>>> AT LINE 018: r : 2.0
>>> AT LINE 018: root = 2.0
>>> AT LINE 039: RETURN FROM root
>>> AT LINE 039: root : 2.0
>>> AT LINE 039: root = 2.0
>>> AT LINE 022
>>> AT LINE 023
The square root of >>> AT LINE 023: number : 4
   4 is >>> AT LINE 023: root : 2.0
  2.000000
>>> AT LINE 039: RETURN FROM print
>>> AT LINE 027: number : 4
>>> AT LINE 028

>>> AT LINE 029
Enter new number (0 to quit): >>> AT LINE 030
0
>>> AT LINE 030: number = 0
>>> AT LINE 032
>>> AT LINE 032: number : 0
>>> AT LINE 032
>>> AT LINE 033
>>> AT LINE 033: CALL print
>>> AT LINE 033: number : 0
>>> AT LINE 033: n = 0
>>> AT LINE 033: root = 0.0
>>> AT LINE 022
>>> AT LINE 023
The square root of >>> AT LINE 023: number : 0
   0 is >>> AT LINE 023: root : 0.0
  0.000000
>>> AT LINE 033: RETURN FROM print
>>> AT LINE 027: number : 0

              34 statements executed.
               0 runtime errors.
            2.84 seconds total execution time.
```

These tracing messages will play critical roles in the interactive source-level debugger, the subject of the next two chapters.

Chapter 13

An Interactive Source-Level Debugger

As you've seen in the past several chapters, an interpreter maintains full control of the runtime environment as it executes a program. Because it works with the source program in its intermediate form, an interpreter remains very close to the original Pascal source program. From a routine's parse tree and its activation record on the runtime stack, the interpreter can obtain the current source line number and the names and values of all the routine's local variables and parameters.

One of the greatest benefits of an interpreter is how well it can support an interactive source-level debugger, an extremely powerful software development tool that enables you to interact directly with the interpreter as it executes a program.

Goals and Approach

The goal for this chapter is to develop a simple yet useful interactive source-level debugger. The debugger will have a command language that enables you to:

- Set breakpoints that pause the execution of the source program at specific line numbers.
- Set watchpoints that monitor the values of specific variables whenever their values are accessed or set as the program runs.
- Single-step the execution of the source program statement-by-statement.
- Display the call stack of the source program, including the chain of procedure and function calls and the names of values of all the local variables of each routine in the call chain.
- Display or set the value of a particular variable.
- Resume execution of the source program.
- Immediately terminate execution of the source program.

The approach is to add debugging capabilities to the structure of the Pascal interpreter. You will make very few changes to the existing code. Once the debugger is in place, you will be able to debug interactively any Pascal program that the interpreter can handle.

Machine-Level vs. Source-Level Debugging

You can broadly categorize debuggers used by programmers during software development: *machine-level* and *source-level*.

Machine-level debuggers allow you to debug a program at a low level that is close to the machine language. With such a debugger, you can execute one machine instruction (or assembly statement) at a time and monitor data moving into and out of the machine registers. Machine-level debuggers are often used to debug compiled programs and may have no knowledge of the statements or variable names of the original source program.

A source-level debugger, on the other hand, allows you to debug at the same level as the high-level language of the source program. It knows about the statements and variable names of the program. When you use such a debugger, you can think in terms of the programming language, not the machine instructions. A Pascal source-level debugger allows you to execute a Pascal program one statement at a time, and you're able to monitor and change the values of the program's variables. You refer to the statements by their source line numbers and to the variables, procedures, and functions by their names; hence, source-level debuggers are also called *symbolic debuggers*.
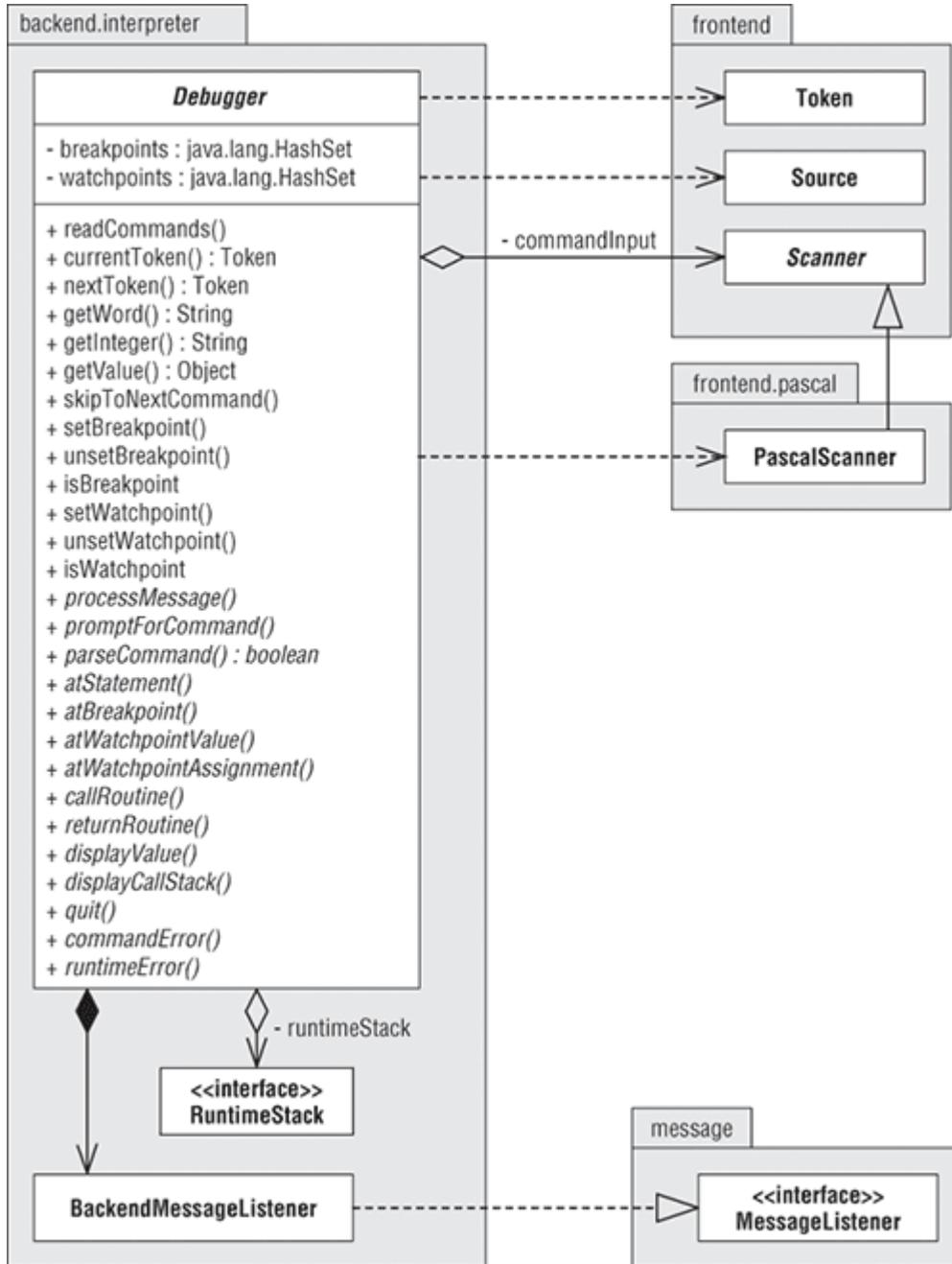
Source-level debuggers are often used with interpreters, although source-level debugging of compiled programs is possible if the compiler generates special debugging information along with the machine instructions.

In the rest of this chapter, you'll develop a simple but useful source-level debugger for the Pascal interpreter. It will be surprisingly straightforward to do so, given the message-passing architecture you've designed.

Debugger Architecture

Figure 13-1 presents the high-level architecture of the debugger. Abstract class `Debugger` in package `backend.interpreter` is the primary interface to all the debugging functionality.

Figure 13-1: Debugger architecture
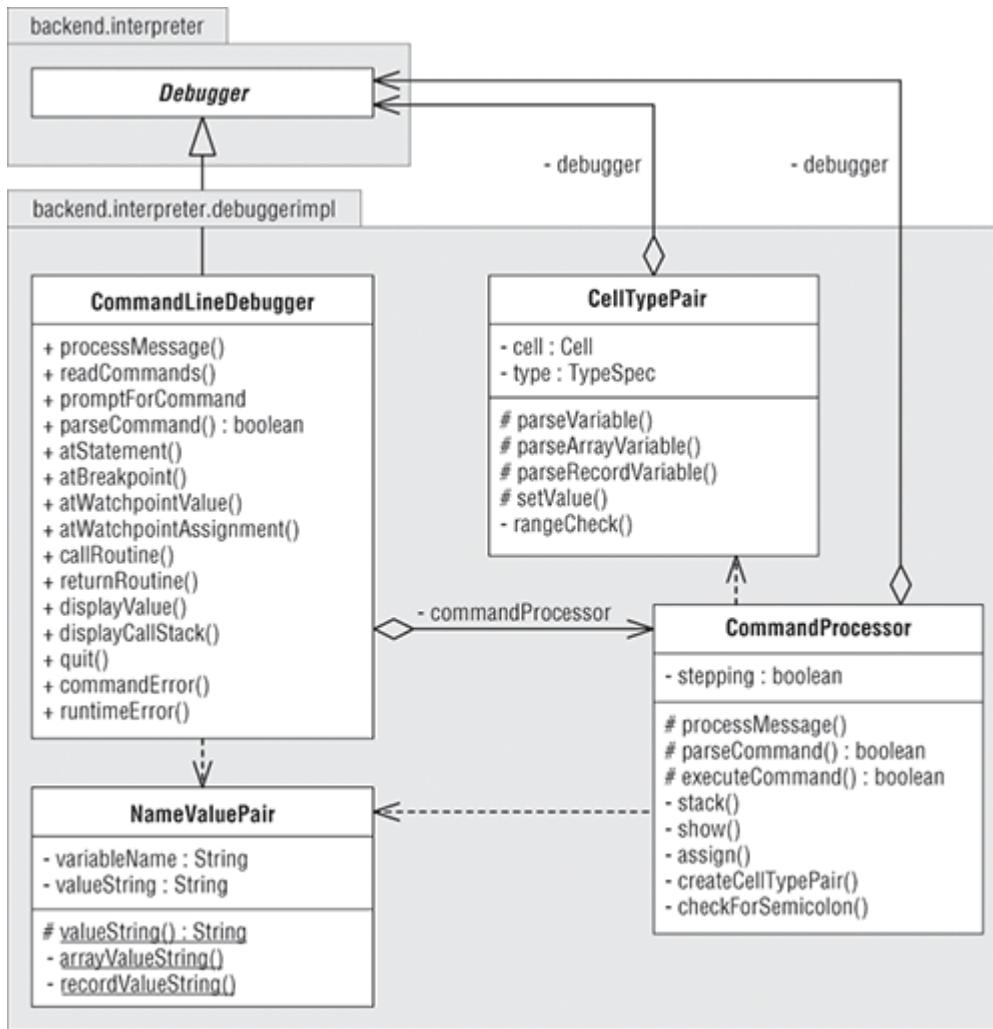
Class `Debugger` keeps track of all the breakpoints and watchpoints with private hash sets. It owns a private reference to the interpreter's runtime stack. The class has numerous methods that comprise the functionality of the debugger, some of which are implemented locally by the class and the rest are abstract methods to be implemented by debugger subclasses.

Because you must be able to interact with the debugger, it needs a command language, and the command language needs a parser. You can see some of the parsing methods in class `Debugger`, such as `currentToken()`, `nextToken()`, `getWord()`, and `getInteger()`. The command-line parser will read tokens from the private `commandInput`, which you'll implement by reusing the `Token`, `Source`, `Scanner` and `PascalScanner` classes from the `frontend` and `frontend.pascal` packages.

The debugger listens to the messages coming from the back end in order to monitor the execution of the source program. It uses a nested listener class that implements the `MessageListener` interface in package `message`.

Figure 13-2 shows the `CommandLineDebugger` subclass of class `Debugger` in package `backend.interpreter.debuggerimpl`.

Figure 13-2: Architecture of the command-line debugger

Class `CommandLineDebugger` implements a debugger that a programmer interacts with by typing commands on a command line. Since it's a subclass of `Debugger`, it implements all of the latter's abstract methods. `CommandLineDebugger` delegates most of the work to class `CommandProcessor`, to which it owns a private reference.

Class `CommandProcessor` parses and executes debugger commands. Its private `stepping` field indicates whether or not you are single-stepping the source program's execution. The class delegates parsing variable names, which may include array subscripts and record fields, to class `CellTypePair`. Class `NameValuePair` associates a variable name with a string representation of its value, which can be a scalar, array, or record. Classes `CommandProcessor` and `CellTypePair` each owns a private reference back to class `Debugger`.

Design Note

By using the abstract class `Debugger` as the interface to the debugger, you can have multiple implementations of the source-level debugger. In this chapter, you implement a command-line version in subclass `CommandLineDebugger`, and in the next chapter you'll implement a GUI-based version in subclass `GUIDebugger`.

Class `CommandProcessor` and the auxiliary classes `CellTypePair` and `NameValuePair` together "factor out" the common debugger functionality. Other debugger implementations can use these classes unchanged.

Listing 13-1 shows the constructor and the inner backend message listener of the abstract class `Debugger`.

Listing 13-1: The constructor and the inner message listener class of class `Debugger`

```
/**
 * <h1>Debugger</h1>
 *
 * <p>Interface for the interactive source-level debugger.</p>
 */
public abstract class Debugger
{
    private RuntimeStack runtimeStack;      // runtime stack
    private HashSet<Integer> breakpoints;   // set of breakpoints
    private HashSet<String> watchpoints;    // set of watchpoints

    private Scanner commandInput;           // input source for commands

    /**
     * Constructor.
     * @param backend the back end.
     * @param runtimeStack the runtime stack.
     */
    public Debugger(Backend backend, RuntimeStack runtimeStack)
    {
        this.runtimeStack = runtimeStack;
        backend.addMessageListener(new BackendMessageListener());

        breakpoints = new HashSet<Integer>();
        watchpoints = new HashSet<String>();

        // Create the command input from the standard input.
        try {
```

```
                commandInput = new PascalScanner(
                                   new Source(
                                       new BufferedReader(
                                           new InputStreamReader(System.in))));
        }
        catch(IOException ignore) {};
    }

    /**
     * Listener for back end messages.
     */
    private class BackendMessageListener implements MessageListener
    {
        /**
         * Called by the back end whenever it produces a message.
         * @param message the message.
         */
        public void messageReceived(Message message)
        {
            processMessage(message);
        }
    }

    ...
}
```

The constructor creates the backend message listener and adds it to the back end's list of message listeners. It creates `commandInput` as a `PascalScanner` whose `Source` is created from the system input file `System.in`.

Method `messageReceived()` of the nested inner class `BackendMessageListener` simply calls `processMessage()` each time it receives a message from the back end. As you'll soon see, this latter method is abstract and will be implemented by a debugger subclass.

You will implement a very basic parser for the debugger. Listing 13-2 shows the `Debugger` methods that read and parse debugger commands.

Listing 13-2: Methods of class `Debugger` that read and parse debugger commands

```
/**
 * Read the debugger commands.
 */
public void readCommands()
{
    do {
        promptForCommand();
    } while (parseCommand());
}

/**
 * Return the current token from the command input.
 * @return the token.
 * @throws Exception if an error occurred.
 */
public Token currentToken()
    throws Exception
{
    return commandInput.currentToken();
}

/**
 * Return the next token from the command input.
 * @return the token.
 * @throws Exception if an error occurred.
 */
public Token nextToken()
    throws Exception
{
    return commandInput.nextToken();
}

/**
 * Get the next word token from the command input.
 * @param errorMessage the error message if an exception is thrown.
 * @return the text of the word token.
 * @throws Exception if an error occurred.
 */
public String getWord(String errorMessage)
    throws Exception
{
    Token token = currentToken();
    TokenType type = token.getType();

    if (type == IDENTIFIER) {
        String word = token.getText().toLowerCase();
        nextToken();
        return word;
    }
    else {
        throw new Exception(errorMessage);
    }
}

/**
 * Get the next integer constant token from the command input.
 * @param errorMessage the error message if an exception is thrown.
 * @return the constant integer value.
 * @throws Exception if an error occurred.
 */
public Integer getInteger(String errorMessage)
    throws Exception
{
    Token token = currentToken();
    TokenType type = token.getType();

    if (type == INTEGER) {
        Integer value = (Integer) token.getValue();
        nextToken();
        return value;
    }
    else {
        throw new Exception(errorMessage);
    }
```

```
    }

    /**
     * Get the next constant value token from the command input.
     * @param errorMessage the error message if an exception is thrown.
     * @return the constant value.
     * @throws Exception if an error occurred.
     */
    public Object getValue(String errorMessage)
        throws Exception
    {
        Token token = currentToken();
        TokenType tokenType = token.getType();
        boolean sign = false;
        boolean minus = false;

        // Unary plus or minus sign.
        if ((tokenType == MINUS) | (tokenType == PLUS)) {
            sign = true;
            minus = tokenType == MINUS;
            token = nextToken();
            tokenType = token.getType();
        }

        switch ((PascalTokenType) tokenType) {

            case INTEGER: {
                Integer value = (Integer) token.getValue();
                nextToken();
                return minus ? -value : value;
            }

            case REAL: {
                Float value = (Float) token.getValue();
                nextToken();
                return minus ? -value : value;
            }

            case STRING: {
                if (sign) {
                    throw new Exception(errorMessage);
                }
                else {
                    String value = (String) token.getValue();
                    nextToken();
                    return value.charAt(0);
                }
            }

            case IDENTIFIER: {
                if (sign) {
                    throw new Exception(errorMessage);
                }
                else {
                    String name = token.getText();
                    nextToken();

                    if (name.equalsIgnoreCase("true")) {
                        return (Boolean) true;
                    }
                    else if (name.equalsIgnoreCase("false")) {
                        return (Boolean) false;
                    }
                    else {
                        throw new Exception(errorMessage);
                    }
                }
            }

            default: {
                throw new Exception(errorMessage);
            }
        }
    }

    /**
     * Skip the rest of this command input line.
     * @throws Exception if an error occurred.
     */
    public void skipToNextCommand()
        throws Exception
    {
        commandInput.skipToNextLine();
    }
```

Method `readCommands()` simply loops to prompt for and parse debugger commands. It calls `promptForCommand()` which you'll see later is an abstract method to be implemented by the debugger subclasses. The loop repeats as long as method `parseCommand()` returns true. This latter method is also abstract.

Methods `currentToken()` and `nextToken()` simply call their scanner counterparts. Methods `getWord()` and `getInteger()` parse and return the next word and integer constant tokens, respectively. Method `getValue()` parses and returns the value of a numeric, string, or boolean value. Method `skipToNextCommand()` skips the rest of the current command line.

Listing 13-3 shows the `Debugger` breakpoint and watchpoint methods. They add and remove items from the `breakpoints` and `watchpoints` hash sets and test whether a statement line number or variable name is in the corresponding hash set.

Listing 13-3: Breakpoint and watchpoint methods of class `Debugger`

```
    /**
     * Set a breakpoint at a source line.
     * @param lineNumber the source line number.
     */
    public void setBreakpoint(Integer lineNumber)
    {
        breakpoints.add(lineNumber);
    }
```

```
/**
 * Remove a breakpoint at a source line.
 * @param lineNumber the source line number.
 */
public void unsetBreakpoint(Integer lineNumber)
{
    breakpoints.remove(lineNumber);
}

/**
 * Check if a source line is at a breakpoint.
 * @param lineNumber the source line number
 * @return true if at a breakpoint, else false.
 */
public boolean isBreakpoint(Integer lineNumber)
{
    return breakpoints.contains(lineNumber);
}

/**
 * Set a watchpoint on a variable.
 * @param name the variable name.
 */
public void setWatchpoint(String name)
{
    watchpoints.add(name);
}

/**
 * Remove a watchpoint on a variable.
 * @param name the variable name.
 */
public void unsetWatchpoint(String name)
{
    watchpoints.remove(name);
}

/**
 * Check if a variable is a watchpoint.
 * @param name the variable name.
 * @return true if a watchpoint, else false.
 */
public boolean isWatchpoint(String name)
{
    return watchpoints.contains(name);
}
```

[Listing 13-4](#) shows the abstract methods of class `Debugger`. Each of these methods will be implemented by the debugger subclasses.

[Listing 13-4:](#) Abstract methods of class `Debugger`

```
/**
 * Process a message from the back end.
 * @param message the message.
 */
public abstract void processMessage(Message message);

/**
 * Display a prompt for a debugger command.
 */
public abstract void promptForCommand();

/**
 * Parse a debugger command.
 * @return true to parse another command immediately, else false.
 */
public abstract boolean parseCommand();

/**
 * Process a source statement.
 * @param lineNumber the statement line number.
 */
public abstract void atStatement(Integer lineNumber);

/**
 * Process a breakpoint at a statement.
 * @param lineNumber the statement line number.
 */
public abstract void atBreakpoint(Integer lineNumber);

/**
 * Process the current value of a watchpoint variable.
 * @param lineNumber the current statement line number.
 * @param name the variable name.
 * @param value the variable&apos;s value.
 */
public abstract void atWatchpointValue(Integer lineNumber,
                                        String name, Object value);

/**
 * Process the assigning a new value to a watchpoint variable.
 * @param lineNumber the current statement line number.
 * @param name the variable name.
 * @param value the new value.
 */
public abstract void atWatchpointAssignment(Integer lineNumber,
                                             String name, Object value);

/**
 * Process calling a declared procedure or function.
 * @param lineNumber the current statement line number.
 * @param name the routine name.
 */
public abstract void callRoutine(Integer lineNumber, String routineName);

/**
 * Process returning from a declared procedure or function.
 * @param lineNumber the current statement line number.
 * @param name the routine name.
 */
public abstract void returnRoutine(Integer lineNumber, String routineName);

/**
 * Display a value.
```

```
 * @param valueString the value string.
 */
public abstract void displayValue(String valueString);

/**
 * Display the call stack.
 * @param stack the list of elements of the call stack.
 */
public abstract void displayCallStack(ArrayList stack);

/**
 * Terminate execution of the source program.
 */
public abstract void quit();

/**
 * Handle a debugger command error.
 * @param errorMessage the error message.
 */
public abstract void commandError(String errorMessage);

/**
 * Handle a source program runtime error.
 * @param errorMessage the error message.
 * @param lineNumber the source line number where the error occurred.
 */
public abstract void runtimeError(String errorMessage, Integer lineNumber);
```

Design Note

What the debugger should do at a statement breakpoint may depend on whether the debugger is command-line- or GUI-based, so each debugger subclass may implement method `atBreakpoint()` differently from the others.

Runtime Data Input vs. Debugger Command Input

The Pascal interpreter has been using the standard input `System.in` to provide runtime input to the executing source program (see class `Executor` in Listing 12-12). But now you also want to read the debugger commands from the standard input (see Listing 13-1). For some Pascal source programs, we may need to separate the two runtime input streams. Therefore, as shown in Listing 13-5, modify the `main()` method of class `Pascal` to accept another optional argument, the path of the runtime input file.

Listing 13-5: A new version of the `main()` method of class Pascal

```
/**
 * The main method.
 * @param args command-line arguments: "compile" or "execute" followed by
 *             optional flags followed by the source file path followed by
 *             an optional runtime input data file path.
 */
public static void main(String args[])
{
    try {
        String operation = args[0];

        // Operation.
        if (!(   operation.equalsIgnoreCase("compile")
              || operation.equalsIgnoreCase("execute"))) {
            throw new Exception();
        }

        int i = 0;
        String flags = "";

        // Flags.
        while ((++i < args.length) && (args[i].charAt(0) == '-')) {
            flags += args[i].substring(1);
        }

        String sourcePath = null;
        String inputPath = null;

        // Source path.
        if (i < args.length) {
            sourcePath = args[i];
        }
        else {
            throw new Exception();
        }

        // Runtime input data file path.
        if (++i < args.length) {
            inputPath = args[i];

            File inputFile = new File(inputPath);
            if (! inputFile.exists()) {
                System.out.println("Input file '" + inputPath +
                                   "' does not exist.");
                throw new Exception();
            }
        }

        new Pascal(operation, sourcePath, inputPath, flags);
    }
    catch (Exception ex) {
        System.out.println(USAGE);
    }
}
```

Method `main()` passes the input data file path to the constructor, which will pass it to a modified version of the `BackendFactory` class's `createBackend()` method.

Listing 13-6 shows a new constructor of the class `Executor` in package `backend.interpreter`, which now creates `standardIn` using a `Source` created from the input data file path, if that file path is not null. Otherwise, it uses `System.in` as before.

Listing 13-6: A new version of the constructor of class `Executor`

```
/**
```

```
 * <h1>Executor</h1>
 *
 * <p>The executor for an interpreter back end.</p>
 *
 * <p>Copyright (c) 2009 by Ronald Mak</p>
 * <p>For instructional purposes only.  No warranties.</p>
 */
public class Executor extends Backend
{
    protected static int executionCount;
    protected static RuntimeStack runtimeStack;
    protected static RuntimeErrorHandler errorHandler;

    protected static Scanner standardIn;       // standard input
    protected static PrintWriter standardOut;  // standard output

    protected Debugger debugger;  // interactive source-level debugger

    static {
        executionCount = 0;
        runtimeStack = MemoryFactory.createRuntimeStack();
        errorHandler = new RuntimeErrorHandler();
        standardOut = new PrintWriter(new PrintStream(System.out));
    }

    /**
     * Constructor.
     */
    public Executor(String inputPath)
    {
        try {
            standardIn = inputPath != null
                             ? new PascalScanner(
                                   new Source(
                                       new BufferedReader(
                                           new FileReader(inputPath))))
                             : new PascalScanner(
                                   new Source(
                                       new BufferedReader(
                                           new InputStreamReader(System.in))));
        }
        catch (IOException ignored) {}

        debugger = BackendFactory.createDebugger(COMMAND_LINE, this,
                                                 runtimeStack);
    }

    ...
}
```

Class `Executor` now has a private reference to a `Debugger` object, which it creates with a call to the `BackendFactory` method `createDebugger()` passing the enumerated value `COMMAND_LINE`, a reference to itself (as a subclass of class `Backend`), and a reference to the runtime stack.

Creating a Command-Line Debugger

shows the new static factory method `createDebugger()` in class `BackendFactory`.

Listing 13-7: Method `createDebugger()` in class `BackendFactory`

```
    /**
     * Create a debugger.
     * @param type the type of debugger (COMMAND_LINE or GUI).
     * @param backend the backend.
     * @param runtimeStack the runtime stack
     * @return the debugger
     */
    public static Debugger createDebugger(DebuggerType type, Backend backend,
                                          RuntimeStack runtimeStack)
    {
        switch (type) {
            case COMMAND_LINE: {
                return new CommandLineDebugger(backend, runtimeStack);
            }

            case GUI: {
                return null;
            }

            default: {
                return null;
            }
        }
    }
```

In this chapter, factory method `createDebugger()` can create only a command-line debugger. It will create a GUI debugger in the next chapter.

shows the enumerated type `DebuggerType`.

Listing 13-8: Enumerated type `DebuggerType`

```
/**
 * <h1>DebuggerType</h1>
 *
 * <p>Debugger types.</p>
 */
public enum DebuggerType
{
    COMMAND_LINE, GUI
}
```

A Simple Command Language

At run time, the command-line debugger prompts you for a command, which you then type and enter. As shown earlier in

, the debugger subclass `CommandLineDebugger` delegates the work of parsing and executing a command to class `CommandProcessor`.

shows class `CommandLineDebugger`. Most of its methods simply print messages to the standard output, and they are called by methods of class `CommandProcessor`. For the command-line debugger, methods `callRoutine()` and `returnRoutine()` do nothing.

Listing 13-9: Class `CommandLineDebugger`

```java
/**
 * <h1>CommandLineDebugger</h1>
 *
 * <p>Command line version of the interactive source-level debugger.</p>
 */
public class CommandLineDebugger extends Debugger
{
    private CommandProcessor commandProcessor;

    /**
     * Constructor.
     * @param backend the back end.
     * @param runtimeStack the runtime stack.
     */
    public CommandLineDebugger(Backend backend, RuntimeStack runtimeStack)
    {
        super(backend, runtimeStack);
        commandProcessor = new CommandProcessor(this);
    }

    /**
     * Process a message from the back end.
     * @param message the message.
     */
    public void processMessage(Message message)
    {
        commandProcessor.processMessage(message);
    }

    /**
     * Display a prompt for a debugger command.
     */
    public void promptForCommand()
    {
        System.out.print(">>> Command? ");
    }

    /**
     * Parse a debugger command.
     * @return true to parse another command immediately, else false.
     */
    public boolean parseCommand()
    {
        return commandProcessor.parseCommand();
    }

    /**
     * Process a source statement.
     * @param lineNumber the statement line number.
     */
    public void atStatement(Integer lineNumber)
    {
        System.out.println("\n>>> At line " + lineNumber);
    }

    /**
     * Process a breakpoint at a statement.
     * @param lineNumber the statement line number.
     */
    public void atBreakpoint(Integer lineNumber)
    {
        System.out.println("\n>>> Breakpoint at line " + lineNumber);
    }

    /**
     * Process the current value of a watchpoint variable.
     * @param lineNumber the current statement line number.
     * @param name the variable name.
     * @param value the variable&apos;s value.
     */
    public void atWatchpointValue(Integer lineNumber,
                                  String name, Object value)
    {
        System.out.println("\n>>> At line " + lineNumber + ": " +
                           name + ": " + value.toString());
    }

    /**
     * Process the assigning a new value to a watchpoint variable.
     * @param lineNumber the current statement line number.
     * @param name the variable name.
     * @param value the new value.
     */
    public void atWatchpointAssignment(Integer lineNumber,
                                       String name, Object value)
    {
        System.out.println("\n>>> At line " + lineNumber + ": " +
                           name + " := " + value.toString());
    }

    /**
     * Process calling a declared procedure or function.
     * @param lineNumber the current statement line number.
     * @param name the routine name.
     */
    public void callRoutine(Integer lineNumber, String routineName) {}

    /**
     * Process returning from a declared procedure or function.
     * @param lineNumber the current statement line number.
     * @param name the routine name.
     */
    public void returnRoutine(Integer lineNumber, String routineName) {}

    /**
     * Display a value.
     * @param valueString the value string.
     */
    public void displayValue(String valueString)
```

```
            {
                System.out.println(valueString);
            }

            /**
             * Display the call stack.
             * @param stack the list of elements of the call stack.
             */
            public void displayCallStack(ArrayList stack)
            {
                for (Object item : stack) {

                    // Name of a procedure or function.
                    if (item instanceof SymTabEntry) {
                        SymTabEntry routineId = (SymTabEntry) item;
                        String routineName = routineId.getName();
                        int level = routineId.getSymTab().getNestingLevel();
                        Definition definition = routineId.getDefinition();

                        System.out.println(level + ": " +
                                        definition.getText().toUpperCase() + " " +
                                        routineName);
                    }

                    // Variable name-value pair.
                    else if (item instanceof NameValuePair) {
                        NameValuePair pair = (NameValuePair) item;
                        System.out.print("  " + pair.getVariableName() + ": ");
                        displayValue(pair.getValueString());
                    }
                }
            }

            /**
             * Terminate execution of the source program.
             */
            public void quit()
            {
                System.out.println("Program terminated.");
                System.exit(-1);
            }

            /**
             * Handle a debugger command error.
             * @param errorMessage the error message.
             */
            public void commandError(String errorMessage)
            {
                System.out.println("!!! ERROR: " + errorMessage);
            }

            /**
             * Handle a source program runtime error.
             * @param errorMessage the error message.
             * @param lineNumber the source line number where the error occurred.
             */
            public void runtimeError(String errorMessage, Integer lineNumber)
            {
                System.out.print("!!! RUNTIME ERROR");
                if (lineNumber != null) {
                    System.out.print(" at line " + String.format("%03d", lineNumber));
                }
                System.out.println(": " + errorMessage);
            }
        }
```

Method `processMessage()` is called by method `messageReceived()` of the `Debugger` class's inner `BackendMessageListener` class (see Listing 13-1). Method `processMessage()` is abstract in the `Debugger` class and is implemented by the `CommandLineDebugger` subclass.

Method `displayCallStack()` is passed an array list that contains a symbol table entry for each name of a procedure or function in the current call stack. Following each symbol table entry are `NameValuePair` entries representing the names and current values of the local variables. An example of what the method can print is

```
1: FUNCTION root
  r: 2.0
  x: 4.0
0: PROGRAM newton
  root: ?
  number: 4
```

where program `newton` at nesting level 0 has called function `root` at nesting level 1, and `r` (value 2.0) and `x` (value 4.0) are local to function `root`, and `root` (undefined value) and `number` (value 4) are local to the program.

Displaying Values

Class `CommandProcessor` delegates displaying values to class `NameValuePair`. Listing 13-10 shows the key methods.

Listing 13-10: Key methods of class `NameValuePair`

```
/**
 * <h1>NameValuePair</h1>
 *
 * <p>Variable name and its value string pair used by the debugger.</p>
 */
public class NameValuePair
{
    private String variableName;  // variable's name
    private String valueString;   // variable's value string

    /**
     * Constructor.
     * @param variableName the variable's name
     * @param value the variable's current value
     */
    protected NameValuePair(String variableName, Object value)
    {
        this.variableName = variableName;
        this.valueString = valueString(value);
    }

    private static final int MAX_DISPLAYED_ELEMENTS = 10;

    /**
```

```java
     * Convert a value into a value string.
     * @param value the value.
     * @return the value string.
     */
    protected static String valueString(Object value)
    {
        StringBuilder buffer = new StringBuilder();

        // Undefined value.
        if (value == null) {
            buffer.append("?");
        }

        // Dereference a VAR parameter.
        else if (value instanceof Cell) {
            buffer.append(valueString(((Cell) value).getValue()));
        }

        // Array value.
        else if (value instanceof Cell[]) {
            arrayValueString((Cell[]) value, buffer);
        }

        // Record value.
        else if (value instanceof HashMap) {
            recordValueString((HashMap) value, buffer);
        }

        // Character value.
        else if (value instanceof Character) {
            buffer.append("&apos;").append((Character) value).append("&apos;");
        }

        // Numeric or boolean value.
        else {
            buffer.append(value.toString());
        }

        return buffer.toString();
    }

    /**
     * Convert an array value into a value string.
     * @param array the array.
     * @param buffer the StringBuilder to use.
     */
    private static void arrayValueString(Cell array[], StringBuilder buffer)
    {
        int elementCount = 0;
        boolean first = true;
        buffer.append("[");

        // Loop over each array element up to MAX_DISPLAYED_ELEMENTS times.
        for (Cell cell : array) {
            if (first) {
                first = false;
            }
            else {
                buffer.append(", ");
            }

            if (++elementCount <= MAX_DISPLAYED_ELEMENTS) {
                buffer.append(valueString(cell.getValue()));
            }
            else {
                buffer.append("...");
                break;
            }
        }

        buffer.append("]");
    }

    /**
     * Convert a record value into a value string.
     * @param array the record.
     * @param buffer the StringBuilder to use.
     */
    private static void recordValueString(HashMap<String, Cell> record,
                                           StringBuilder buffer)
    {
        boolean first = true;
        buffer.append("{");

        Set<Map.Entry<String, Cell>> entries = record.entrySet();
        Iterator<Map.Entry<String, Cell>> it = entries.iterator();

        // Loop over each record field.
        while (it.hasNext()) {
            Map.Entry<String, Cell> entry = it.next();

            if (first) {
                first = false;
            }
            else {
                buffer.append(", ");
            }

            buffer.append(entry.getKey()).append(": ")
                  .append(valueString(entry.getValue().getValue()));
        }

        buffer.append("}");
    }
}
```

Method `valueString()` builds and returns the string representation of a value. For an undefined value, it returns `"?"` and for a scalar value, it returns the `toString()` of the value. The method calls `arrayValueString()` for an array value or `recordValueString()` for a record value.

Method `arrayValueString()` loops over the elements of an array value and generates a comma-separated list of element values surrounded by square brackets, up to a maximum of ten elements. The method calls `valueString()` recursively for each element value. An example array value string is

```
"[&apos;p&apos;, &apos;r&apos;, &apos;o&apos;, &apos;g&apos;, &apos;r&apos;, &apos;a&apos;, &apos;m&apos;, &apos; &apos;, &apos; &apos;, &apos; &apos;, ...]"
```

Method `recordValueString()` loops over the fields of a record value and generates a comma-separated list of field-name : value-string pairs surrounded by curly brackets. The method calls `valueString()` recursively for each field value. An example record value string is

```
"{nextindex: 0, number: 1}"
```

A value can be a composition of array elements and record fields. An example value string for such a value is

```
"[{word: [&apos;p&apos;, &apos;r&apos;, &apos;o&apos;, &apos;g&apos;, &apos;r&apos;, &apos;a&apos;, &apos;m&apos;, &apos; &apos;, &apos; &apos;, &apos; &apos;, ...],
  firstnumberindex: 1, lastnumberindex: 1},
 {word: [?, ?, ?, ?, ?, ?, ?, ?, ?, ...], firstnumberindex: ?,
  lastnumberindex: ?}]"
```

Parsing Variable Names

Class `CommandProcessor` delegates parsing variable names in a debugger command to class `CellTypePair`. shows the parsing methods.

Listing 13-11: Parsing methods of class `CellTypePair`

```java
/**
 * <h1>CellTypePair</h1>
 *
 * <p>Memory cell and data type pair used by the debugger.</p>
 */
public class CellTypePair
{
    private Cell cell;          // memory cell
    private TypeSpec type;      // data type
    private Debugger debugger;  // parent debugger

    /**
     * Constructor.
     * @param type the data type.
     * @param cell the memory cell.
     * @param debugger the parent debugger.
     * @throws Exception if an error occurred.
     */
    protected CellTypePair(TypeSpec type, Cell cell, Debugger debugger)
        throws Exception
    {
        this.type = type;
        this.cell = cell;
        this.debugger = debugger;

        parseVariable();
    }

    // Synchronization set for variable modifiers.
    protected static final EnumSet<PascalTokenType> MODIFIER_SET =
        EnumSet.of(LEFT_BRACKET, DOT);

    /**
     * Parse a variable in the command to obtain its memory cell.
     * @param type the variable&apos;s data type.
     * @param cell the variable&apos;s memory cell.
     * @throws Exception if an error occurred.
     */
    protected void parseVariable()
        throws Exception
    {
        TypeForm form = type.getForm();
        Object value = cell.getValue();

        // Loop to process array subscripts and record fields.
        while (MODIFIER_SET.contains(debugger.currentToken().getType())) {
            if (form == TypeFormImpl.ARRAY) {
                parseArrayVariable((Cell[]) value);
            }
            else if (form == TypeFormImpl.RECORD) {
                parseRecordVariable((HashMap) value);
            }

            value = cell.getValue();
            form = type.getForm();
        }
    }

    /**
     * Parse an array variable.
     * @param array the array variable.
     * @throws Exception if an error occurred.
     */
    private void parseArrayVariable(Cell array[])
        throws Exception
    {
        debugger.nextToken();

        int index = debugger.getInteger("Integer index expected.");
        int minValue = 0;
        TypeSpec indexType = (TypeSpec) type.getAttribute(ARRAY_INDEX_TYPE);

        rangeCheck(index, indexType, "Index out of range.");
        type = (TypeSpec) type.getAttribute(ARRAY_ELEMENT_TYPE);

        if (indexType.getForm() == SUBRANGE) {
            minValue = (Integer) indexType.getAttribute(SUBRANGE_MIN_VALUE);
        }

        cell = array[index - minValue];

        if (debugger.currentToken().getType() == RIGHT_BRACKET) {
            debugger.nextToken();
        }
        else {
            throw new Exception("] expected.");
        }
    }

    /**
     * Parse a record variable.
     * @param record the record variable.
     * @throws Exception if an error occurred.
     */
```

```
    private void parseRecordVariable(HashMap record)
        throws Exception
    {
        debugger.nextToken();

        String fieldName = debugger.getWord("Field name expected.");

        if (record.containsKey(fieldName)) {
            cell = (Cell) record.get(fieldName);
        }
        else {
            throw new Exception("Invalid field name.");
        }

        SymTab symTab = (SymTab) type.getAttribute(RECORD_SYMTAB);
        SymTabEntry id = symTab.lookup(fieldName);
        type = id.getTypeSpec();
    }

    /**
     * Do a range check on an integer value.
     * @param value the value.
     * @param type the data type.
     * @param errorMessage the error message if an exception is thrown.
     * @throws Exception if an error occurs.
     */
    private void rangeCheck(int value, TypeSpec type, String errorMessage)
        throws Exception
    {
        TypeForm form = type.getForm();
        Integer minValue = null;
        Integer maxValue = null;

        if (form == SUBRANGE) {
            minValue = (Integer) type.getAttribute(SUBRANGE_MIN_VALUE);
            maxValue = (Integer) type.getAttribute(SUBRANGE_MAX_VALUE);
        }
        else if (form == ENUMERATION) {
            ArrayList<SymTabEntry> constants = (ArrayList<SymTabEntry>)
                                    type.getAttribute(ENUMERATION_CONSTANTS);
            minValue = 0;
            maxValue = constants.size() - 1;
        }

        if ((minValue != null) && ((value < minValue) || (value > maxValue))) {
            throw new Exception(errorMessage);
        }
    }

    ...
}
```

If method `parseVariable()` succeeds, it sets `cell` to the variable's memory cell, which it found within the memory map of the appropriate activation record on the runtime stack. It also sets `type` to the variable's data type specification object.

The method loops to process array subscripts and record fields, calling `parseArrayVariable()` or `parseRecordVariable()`, respectively. Method `parseArrayVariable()` parses a subscript value, calls `rangeCheck()`, and sets `cell` to the proper element in the memory cell array and `type` to the element type. Method `parseRecordVariable()` sets `cell` to the proper memory cell for the field in the hash map and `type` to the field type.

Method `rangeCheck()` does a range check of a subrange value, whether for a subscript value or, as you'll see shortly, for an assignment.

Class `CellTypePair` also has a method to set a new value into a memory cell, as shown in Listing 13-12.

Listing 13-12: Method `setValue()` of class `CellTypePair`

```
    /**
     * Set the value of the cell.
     * @param value the value.
     * @throws Exception if an error occurred.
     */
    protected void setValue(Object value)
        throws Exception
    {
        if (   (    (type.baseType() == Predefined.integerType)
                 && (value instanceof Integer))
            || (    (type == Predefined.realType)
                 && (value instanceof Float))
            || (    (type == Predefined.booleanType)
                 && (value instanceof Boolean))
            || (    (type == Predefined.charType)
                 && (value instanceof Character)))
        {
            if (type.baseType() == Predefined.integerType) {
                rangeCheck((Integer) value, type, "Value out of range.");
            }

            cell.setValue(value);
        }
        else {
            throw new Exception("Type mismatch.");
        }
    }
```

Method `setValue()` sets the value of cell after verifying type compatibility and, for a subrange type, calling `rangeCheck()`.

Executing Commands

Now that all the pieces are in place, you can examine class `CommandProcessor`, which does the command processing for the source-level debuggers. Its methods make numerous "callbacks" to `Debugger` methods, some of which are implemented by class `CommandLineDebugger`. Listing 13-13 shows the `processMessage()` method.

Listing 13-13: Method `processMessage()` of class `CommandProcessor`

```
/**
 * <h1>CommandProcessor</h1>
 *
 * <p>Command processor for the interactive source-level debugger.</p>
 */
```

```java
public class CommandProcessor
{
    private Debugger debugger;  // the debugger
    private boolean stepping;   // true when single stepping

    ...

    /**
     * Process a message from the back end.
     * @param message the message.
     */
    protected void processMessage(Message message)
    {
        MessageType type = message.getType();

        switch (type) {

            case SOURCE_LINE: {
                int lineNumber = (Integer) message.getBody();

                if (stepping) {
                    debugger.atStatement(lineNumber);
                    debugger.readCommands();
                }
                else if (debugger.isBreakpoint(lineNumber)) {
                    debugger.atBreakpoint(lineNumber);
                    debugger.readCommands();
                }

                break;
            }

            case FETCH: {
                Object body[] = (Object[]) message.getBody();
                String variableName = ((String) body[1]).toLowerCase();

                if (debugger.isWatchpoint(variableName)) {
                    int lineNumber = (Integer) body[0];
                    Object value = body[2];

                    debugger.atWatchpointValue(lineNumber, variableName, value);
                }

                break;
            }

            case ASSIGN: {
                Object body[] = (Object[]) message.getBody();
                String variableName = ((String) body[1]).toLowerCase();

                if (debugger.isWatchpoint(variableName)) {
                    int lineNumber = (Integer) body[0];
                    Object value = body[2];

                    debugger.atWatchpointAssignment(lineNumber, variableName,
                                                    value);
                }

                break;
            }

            case CALL: {
                Object body[] = (Object[]) message.getBody();
                int lineNumber = (Integer) body[0];
                String routineName = (String) body[1];

                debugger.callRoutine(lineNumber, routineName);
                break;
            }

            case RETURN: {
                Object body[] = (Object[]) message.getBody();
                int lineNumber = (Integer) body[0];
                String routineName = (String) body[1];

                debugger.returnRoutine(lineNumber, routineName);
                break;
            }

            case RUNTIME_ERROR: {
                Object body[] = (Object []) message.getBody();
                String errorMessage = (String) body[0];
                Integer lineNumber = (Integer) body[1];

                debugger.runtimeError(errorMessage, lineNumber);
                break;
            }
        }
    }

    ...
}
```

Method `processMessage()` is called by a debugger implementation, such as from the `processMessage()` method of class `CommandLineDebugger` (see). The call chain begins in class `Debugger` from the `messageReceived()` method of the back end message listener (see). In other words, the `processMessage()` method of class `CommandProcessor` processes each runtime message from the back end.

The method processes several types of back end messages, all of which you've seen before with the interpreter:

Message type	Sent by the backend …
SOURCE_LINE	Just before executing each statement.
FETCH	Whenever any variable's value is accessed.
ASSIGN	Whenever any variable's value is set.
CALL	Whenever a procedure or function is called.
RETURN	Upon returning from a procedure or function.
RUNTIME_ERROR	Whenever a runtime error occurs.

These messages enable the debugger to monitor the interpreter as it executes the source program.

shows the `CommandProcessor` class's `parseCommand()` and `executeCommand()` methods.

```
/**
 * Parse a debugger command.
 * @return true to parse another command immediately, else false.
 */
protected boolean parseCommand()
{
    boolean anotherCommand = true;

    // Parse a command.
    try {
        debugger.nextToken();
        String command = debugger.getWord("Command expected.");
        anotherCommand = executeCommand(command);
    }
    catch (Exception ex) {
        debugger.commandError(ex.getMessage());
    }

    // Skip to the next command.
    try {
        debugger.skipToNextCommand();
    }
    catch (Exception ex) {
        debugger.commandError(ex.getMessage());
    }

    return anotherCommand;
}

/**
 * Execute a debugger command.
 * @param command the command string.
 * @return true to parse another command immediately, else false.
 * @throws Exception if an error occurred.
 */
private boolean executeCommand(String command)
    throws Exception
{
    stepping = false;

    if (command.equals("step")) {
        stepping = true;
        checkForSemicolon();
        return false;
    }

    if (command.equals("break")) {
        Integer lineNumber = debugger.getInteger("Line number expected.");
        checkForSemicolon();
        debugger.setBreakpoint((Integer) lineNumber);
        return true;
    }

    if (command.equals("unbreak")) {
        Integer lineNumber = debugger.getInteger("Line number expected.");
        checkForSemicolon();
        debugger.unsetBreakpoint((Integer) lineNumber);
        return true;
    }

    if (command.equals("watch")) {
        String name = debugger.getWord("Variable name expected.");
        checkForSemicolon();
        debugger.setWatchpoint(name);
        return true;
    }

    if (command.equals("unwatch")) {
        String name = debugger.getWord("Variable name expected.");
        checkForSemicolon();
        debugger.unsetWatchpoint(name);
        return true;
    }

    if (command.equals("stack")) {
        checkForSemicolon();
        stack();
        return true;
    }

    if (command.equals("show")) {
        show();
        return true;
    }

    if (command.equals("assign")) {
        assign();
        return true;
    }

    if (command.equals("go")) {
        checkForSemicolon();
        return false;
    }

    if (command.equals("quit")) {
        checkForSemicolon();
        debugger.quit();
    }

    throw new Exception("Invalid command: &apos;" + command + "&apos;.");
}
```

Method `parseCommand()` is called by the `parseCommand()` method of the debugger implementation (again, see Listing 13-9) which is called by the `Debugger` class's `readCommands()` method (see Listing 13-2). Method `parseCommand()` returns true or false to indicate to the loop in method `readCommands()` whether or not to prompt for and read another debugger command.

Method `executeCommand()` executes each debugger command. It executes many of the commands simply by making the appropriate callbacks to the `Debugger` methods.

Listing 13-15 shows the remaining methods of class `CommandProcessor`. Method `executeCommand()` calls these private helper methods.

```java
/**
 * Create the call stack for display.
 */
private void stack()
{
    ArrayList callStack = new ArrayList();

    // Loop over the activation records on the runtime stack
    // starting at the top of stack.
    RuntimeStack runtimeStack = debugger.getRuntimeStack();
    ArrayList<ActivationRecord> arList = runtimeStack.records();
    for (int i = arList.size() - 1; i >= 0; --i) {
        ActivationRecord ar = arList.get(i);
        SymTabEntry routineId = ar.getRoutineId();

        // Add the symbol table entry of the procedure or function.
        callStack.add(routineId);

        // Create and add a name-value pair for each local variable.
        for (String name : ar.getAllNames()) {
            Object value = ar.getCell(name).getValue();
            callStack.add(new NameValuePair(name, value));
        }
    }

    // Display the call stack.
    debugger.displayCallStack(callStack);
}

/**
 * Show the current value of a variable.
 * @throws Exception if an error occurred.
 */
private void show()
    throws Exception
{
    CellTypePair pair = createCellTypePair();
    Cell cell = pair.getCell();

    checkForSemicolon();
    debugger.displayValue(NameValuePair.valueString(cell.getValue()));
}

/**
 * Assign a new value to a variable.
 * @throws Exception if an error occurred.
 */
private void assign()
    throws Exception
{
    CellTypePair pair = createCellTypePair();
    Object newValue = debugger.getValue("Invalid value.");

    checkForSemicolon();
    pair.setValue(newValue);
}

/**
 * Create a cell-data type pair.
 * @return the CellTypePair object.
 * @throws Exception if an error occurred.
 */
private CellTypePair createCellTypePair()
    throws Exception
{
    RuntimeStack runtimeStack = debugger.getRuntimeStack();
    int currentLevel = runtimeStack.currentNestingLevel();
    ActivationRecord ar = null;
    Cell cell = null;

    // Parse the variable name.
    String variableName = debugger.getWord("Variable name expected.");

    // Find the variable&apos;s cell in the call stack.
    for (int level = currentLevel; (cell == null) && (level > 0); --level) {
        ar = runtimeStack.getTopmost(level);
        cell = ar.getCell(variableName);
    }

    if (cell == null) {
        throw new Exception("Undeclared variable name &apos;" + variableName +
                            "&apos;.");
    }

    // VAR parameter.
    if (cell.getValue() instanceof Cell) {
        cell = (Cell) cell.getValue();
    }

    // Find the variable&apos;s symbol table entry.
    SymTab symTab = (SymTab) ar.getRoutineId().getAttribute(ROUTINE_SYMTAB);
    SymTabEntry id = symTab.lookup(variableName);

    return new CellTypePair(id.getTypeSpec(), cell, debugger);
}

/**
 * Verify that a command ends with a semicolon.
 * @throws Exception if an error occurred.
 */
private void checkForSemicolon()
    throws Exception
{
    if (debugger.currentToken().getType() != SEMICOLON) {
        throw new Exception("Invalid command syntax.");
    }
}
```

Method `stack()` creates the call stack as an array list. It obtains the call chain of procedures and functions from the runtime stack and adds the symbol table entry of each routine name to the array list. For each routine, the method accesses all the local names from the activation record on the runtime stack and adds a `NameValuePair` object for each name to the array list. It then passes the array list in a call to `debugger.displayCallStack()`.

Method `show()` displays the current value of a variable. It calls `createCellTypePair()` which creates a `CellTypePair` object. It calls `NameValuePair.valueString()` on the pair's cell value and passes the value string to the debugger's `displayValue()` method.

Method `assign()` allows you to set the value of a variable at run time. It also calls `createCellTypePair()` to create a `CellTypePair` object. It calls the debugger's `getValue()` method to parse a constant value, and then it calls the pair's `setValue()` method to set the value into the pair's cell.

Method `createCellTypePair()` parses a variable name that appears in a debugger command. It searches the activation records on the runtime stack to find the one containing the variable's memory cell. From that activation record, it obtains the corresponding routine's symbol table, from which it accesses the variable's symbol table entry. Finally, the method uses the variable's type specification and its memory cell to create and return a new `CellTypePair` object. As discussed earlier, constructing a `CellTypePair` object will parse any array subscripts and record fields.

Method `checkForSemicolon()` verifies that each command in the simple debugger command language ends with a semicolon.

Breakpoints

The `break` command sets a breakpoint at a statement line number, and the `unbreak` command removes a breakpoint at a statement line number. Examples:

```
>>> Command? break 199;
>>> Command? unbreak 199;
```

When the interpreter is executing a program and reaches a statement that has a breakpoint set, it pauses program execution and the command-line debugger prompts for a command:

```
>>> Breakpoint at line 199
>>> Command?
```

After either of these commands, the command-line debugger prompts for another command.

Watchpoints

The `watch` command sets a watchpoint on a variable name, and the `unwatch` command removes a watchpoint from a variable name. Examples:

```
>>> Command? watch linenumber;
>>> Command? unwatch linenumber;
```

Whenever the interpreter is executing a program and it fetches or sets a variable's value and that variable has a watchpoint set, the command line-debugger displays a message without pausing program execution:

```
>>> At line 61: linenumber := 3

>>> At line 62: linenumber: 3
```

Watchpoints in the simple debugger are very basic. The granularity of a watchpoint is a variable without any array subscripts or record fields. If, for example, a watchpoint is placed on an array variable, then the debugger will display a message if the running source program fetches or sets the value of *any* element of the array.

Single Stepping, Resuming Execution, and Terminating

The `step` command causes the interpreter to execute the current statement and then the command-line debugger prompts for another command. Therefore, by entering successive `step` commands, you can execute a program statement by statement. For example:

```
>>> Breakpoint at line 199
>>> Command? step;

>>> At line 267
>>> Command? step;

>>> At line 271
>>> Command? step;

>>> At line 96
>>> Command?
```

The `go` command causes the interpreter to resume execution of the program, and the `quit` command causes the interpreter to terminate program execution immediately.

Displaying the Call Stack

The `stack` command displays the current call stack.

```
>>> Command? stack;
```

As shown earlier, the command-line debugger displays the names of the procedures and functions in the call chain, and the local names and values of each routine.

Displaying and Setting a Variable's Value

The `show` command displays the current value of a variable, which can have array subscripts and record fields. Examples:

```
>>> Command? show wordtable;
[{word: ['p', 'r', 'o', 'g', 'r', 'a', 'm', ' ', ' ', ' ', ...],
firstnumberindex: 1, lastnumberindex: 1}, {word: ['n', 'e', 'w', 't', 'o', 'n',
' ', ' ', ' ', ' ', ' ', ...], firstnumberindex: 2, lastnumberindex: 2}, {word:
['c', 'o', 'n', 's', 't', ' ', ' ', ' ', ' ', ' ', ' ', ...], firstnumberindex: 3,
lastnumberindex: 3}, {word: ['e', 'p', 's', 'i', 'l', 'o', 'n', ' ', ' ', ' ',
' ', ...], firstnumberindex: 4, lastnumberindex: 4}, {word: ['e', ' ', ' ', ' ',
' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ...], firstnumberindex: 5,
lastnumberindex: 5}, {word: [?, ?, ?, ?, ?, ?, ?, ?, ?, ...],
firstnumberindex: ?, lastnumberindex: ?}, {word: [?, ?, ?, ?, ?, ?, ?,
?, ?, ...], firstnumberindex: ?, lastnumberindex: ?}, {word: [?, ?, ?, ?,
?, ?, ?, ?, ...], firstnumberindex: ?, lastnumberindex: ?},
{word: [?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ...], firstnumberindex: ?,
lastnumberindex: ?}, {word: [?, ?, ?, ?, ?, ?, ?, ?, ?, ...],
```

```
firstnumberindex: ?, lastnumberindex: ?}, ...]

>>> Command? show wordtable[1];
{word: ['p', 'r', 'o', 'g', 'r', 'a', 'm', ' ', ' ', ' ', ' ', ...],
firstnumberindex: 1, lastnumberindex: 1}

>>> Command? show wordtable[1].word;
['p', 'r', 'o', 'g', 'r', 'a', 'm', ' ', ' ', ' ', ' ', ...]

>>> Command? show wordtable[1].word[4];
'g'

>>> Command? show wordtable[2].firstnumberindex;
2
```

The `assign` command allows you to set the value of a variable, which may have array subscripts and record fields. The value must be scalar. For example:

```
>>> Command? assign wordtable[1].word[4] 'x';

>>> Command? show wordtable[1].word;
['p', 'r', 'o', 'x', 'r', 'a', 'm', ' ', ' ', ' ', ' ', ...]
```

As these examples demonstrate, the command-line debugger prompts for another command after a `show` or `assign` command.

Program 13-1: Command-Line Source-Level Debugger

This chapter concludes with a simple debugging session, shown in
.

Listing 13-16: A debugging session

```
001 PROGRAM newton;
002
003 CONST
004     epsilon = 1e-6;
005
006 VAR
007     number : integer;
008
009 FUNCTION root(x : real) : real;
010     VAR
011         r : real;
012
013     BEGIN
014         r := 1;
015         REPEAT
016             r := (x/r + r)/2;
017         UNTIL abs(x/sqr(r) - 1) < epsilon;
018         root := r;
019     END;
020
021 PROCEDURE print(n : integer; root : real);
022     BEGIN
023         writeln('The square root of ', number:4, ' is ', root:8:4);
024     END;
025
026 BEGIN
027     REPEAT
028         writeln;
029         write('Enter new number (0 to quit): ');
030         read(number);
031
032         IF number = 0 THEN BEGIN
033             print(number, 0.0);
034         END
035         ELSE IF number < 0 THEN BEGIN
036             writeln('*** ERROR:  number < 0');
037         END
038         ELSE BEGIN
039             print(number, root(number));
040         END
041     UNTIL number = 0
042 END.

               42 source lines.
                0 syntax errors.
             0.09 seconds total parsing time.

>>> At line 26
>>> Command? break 14;
>>> Command? break 18;
>>> Command? go;

Enter new number (0 to quit): 4

>>> Breakpoint at line 14
>>> Command? stack;
1: FUNCTION root
  r: ?
  x: 4.0
0: PROGRAM newton
  root: ?
  number: 4
>>> Command? watch r;
>>> Command? go;

>>> At line 14: r := 1

>>> At line 16: r: 1.0

>>> At line 16: r: 1.0

>>> At line 16: r := 2.5
```

```
>>> At line 15: r: 2.5

>>> At line 16: r: 2.5

>>> At line 16: r: 2.5

>>> At line 16: r := 2.05

>>> At line 15: r: 2.05

>>> At line 16: r: 2.05

>>> At line 16: r: 2.05

>>> At line 16: r := 2.0006099

>>> At line 15: r: 2.0006099

>>> At line 16: r: 2.0006099

>>> At line 16: r: 2.0006099

>>> At line 16: r := 2.0

>>> At line 15: r: 2.0

>>> Breakpoint at line 18
>>> Command? show r;
2.0
>>> Command? assign r -88;
!!! ERROR: Type mismatch.
>>> Command? assign r -88.8;
>>> Command? go;

>>> At line 18: r: -88.8
The square root of    4 is -88.8000

Enter new number (0 to quit): 16

>>> Breakpoint at line 14
>>> Command? step;

>>> At line 15
>>> Command? step;

>>> At line 16
>>> Command? step;

>>> At line 16
>>> Command? quit;
Program terminated.
```

This is a very simple command-line debugger, but it was relatively easy to develop with the architecture of the interpreter. Of course, much more can be done to add more features to the debugger and enrich its command language.

In the next chapter, you'll develop a source-level debugger with a simple graphical user interface (GUI).

Chapter 14

Framework II: An Integrated Development Environment (IDE)

In the previous chapter, you developed an interactive source-level debugger that enabled you use the command line to interact with the Pascal interpreter as it was running a source program. Even though it was relatively simple to implement and somewhat primitive in its capabilities, it demonstrated how a debugger can be a powerful tool for software developers.

In this chapter, you'll create an even more powerful tool: an interactive development environment, or IDE. This IDE will have a graphical user interface (GUI) comprising several windows: an edit window, a debug window, a call stack window, and a console window. The edit window will have a screen editor where you can edit a Pascal source program and correct any syntax errors. The other windows will come to life as the source program is executing. The debug window will allow you to set breakpoints by clicking on source lines. By repeatedly clicking a button, you'll single-step the source program statement-by-statement. The debug window will also "auto step" by slowly executing the program and continually moving the source line highlight to the current line. The call stack window will show the call stack and the local variables as a tree, and it will update the tree as the source program runs. You'll be able to modify the values of variables at run time. The console window will display the program's output and allow you to enter runtime input data.

The IDE you'll develop in this chapter will have far fewer features than a full-function IDE such as the popular open-source Eclipse environment, but it should give you ideas for what you can accomplish with the code you've already written.

Goals and Approach

The goal for this chapter is to develop a simple GUI-based IDE for Pascal that will have an edit window, a debug window, a call stack window, and a console window.

The approach is to wrap a graphical user interface around the command-line-based debugger you developed in the previous chapter. The GUI code will use multithreading and graphical components of the Java Foundation Classes (JFC, also known informally as Swing).

Multithreaded programming and the Swing classes are beyond the scope of this book. Therefore, this chapter will present the framework of the IDE but not all the programming details.
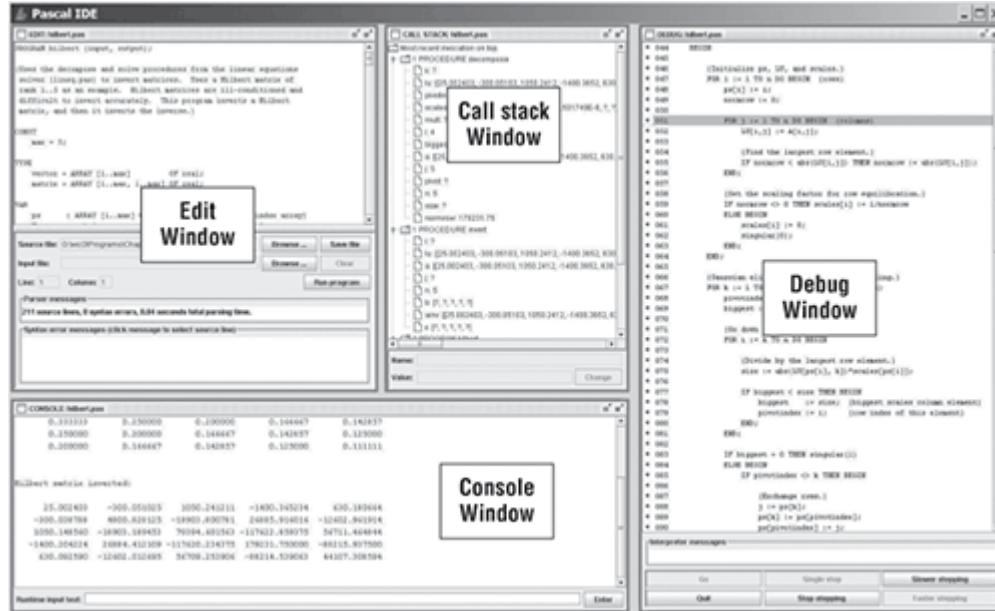
The Pascal IDE

The graphical user interface of the Pascal IDE consists of a main window that contains the edit, debug, call stack, and console windows. You can move, resize, and iconify the main window, and you can also move, resize, and iconify each of the internal windows within the main window.

The IDE Window

Figure 14-1 shows the main Pascal IDE window and its four internal windows.
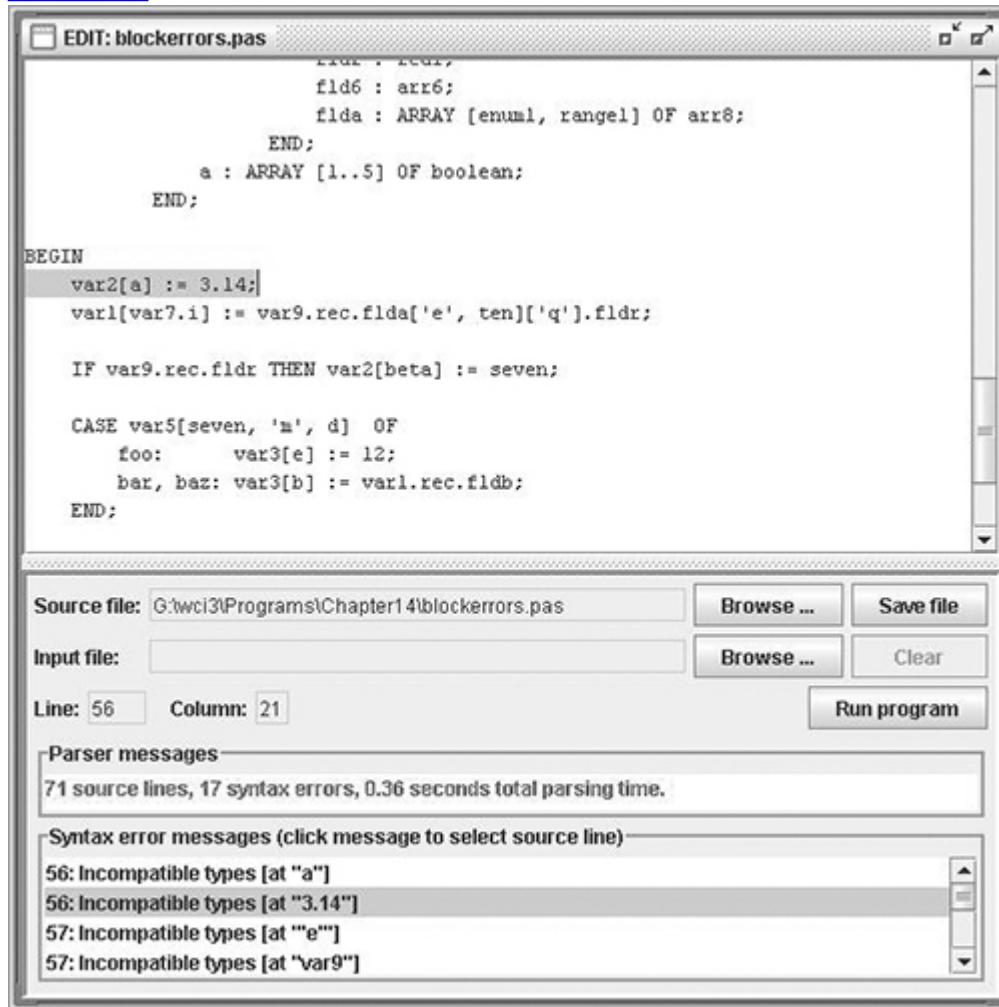
Figure 14-1: The main Pascal IDE window and its internal windows

The Edit Window

Figure 14-2 shows the edit window.

shows the edit window.

Figure 14-2: The edit window

You can edit the Pascal source program in the scrollable text area. The control panel allows you to browse the file system and select the Pascal source file, and for programs that expect to read an input text file at runtime, you can browse for and select the input file.

When you press the Run program button, the Pascal debugger will start. If the parser finds any syntax errors, the error messages are displayed in the Syntax error messages list. If you click on a message, the offending source line becomes highlighted in the text area. You can see this in Figure 14-2.

The Debug Window

If there were no syntax errors after you press the Run program button, the debugger takes over. Figure 14-3 shows the debug window.

Figure 14-3: The debug window

The main scrollable listing area displays the source program listing and it highlights the first executable statement. You can perform several actions in the debug window:
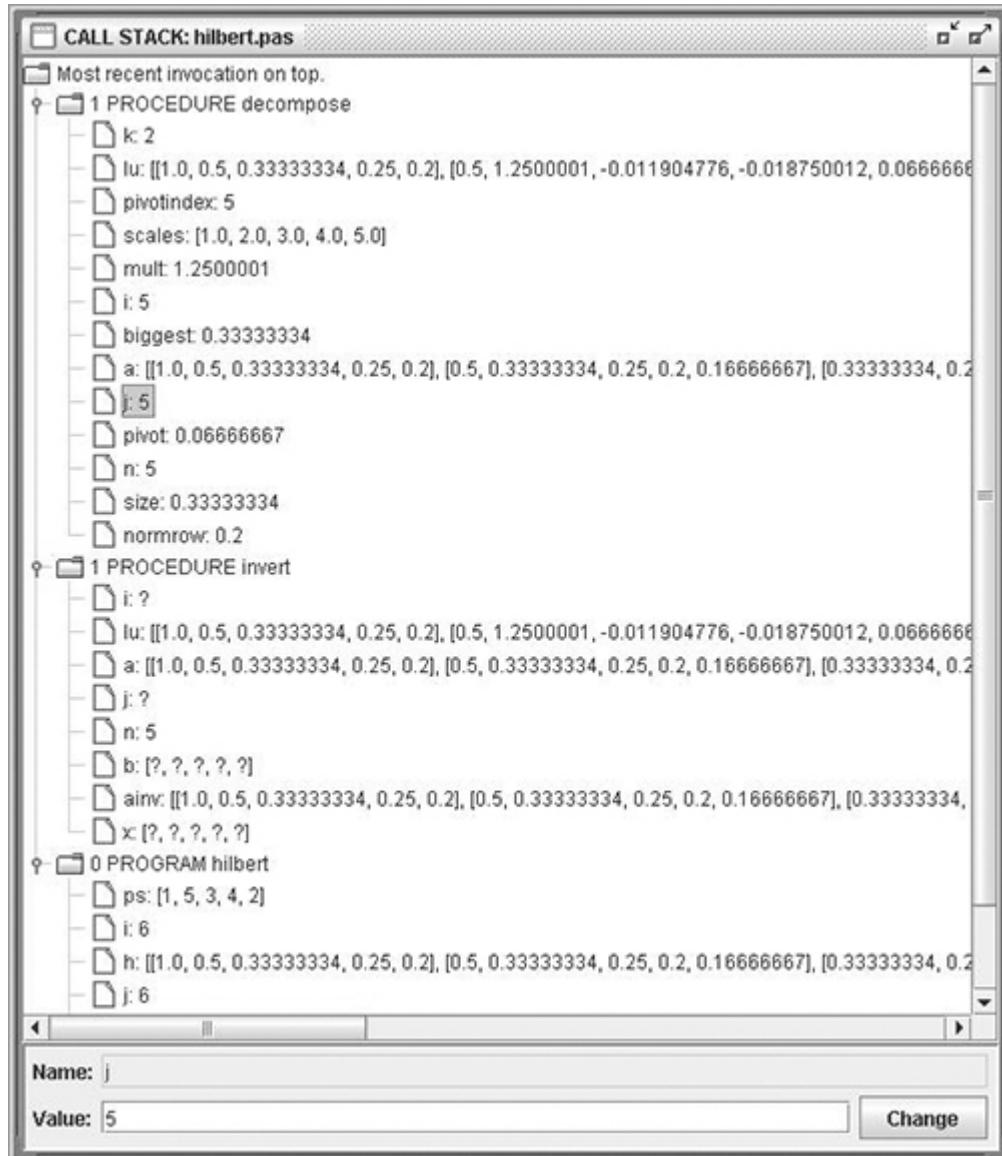
- Press the Go button and the debugger will proceed to execute the source program normally and display the program's runtime output in the console window.
- Press the Single step button and the debugger will execute only the current statement and then move the listing line highlight to the statement it will execute next.
- Press the Auto step button and the debugger will repeatedly perform the single-step operation. The effect is to execute the program slowly while continually moving the listing line highlight to the current line. Press the Slower stepping or the Faster stepping button to tell the debugger to execute the program slower or faster, respectively.
- You can set or unset a breakpoint by clicking on the listing line. In Figure 14-3, there are breakpoints on lines 181 and 192. After you press the Go button or the Auto step button, the debugger will execute the program until it hits a breakpoint.
- Press the Quit button to tell the debugger to terminate the execution of the source program immediately.

The debug window also displays any messages from the Pascal interpreter.

The Call Stack Window

Figure 14-4 shows the call stack window.

Figure 14-4: The call stack window

Figure 14-4: The call stack window

The call stack window displays the current call stack as a tree. Each non-terminal tree node represents the invocation of a routine, either the main program (at the bottom of the tree) or a procedure or function (the most recent on top). The leaf nodes represent local variables and parameters and their values.

If you select a scalar variable in the tree, such as variable ⱼ in Figure 14-4, you can change its value by entering a new value in the Value text field under the tree and pressing the Change button.

The call stack window updates the tree when you single step or auto step the source program execution.

Design Note

The call stack window has only rudimentary functionality. You can only change a scalar value. An obvious improvement is to display the values of array elements and record fields hierarchically as individual child nodes.

The Console Window

Figure 14-5 shows the console window.

Figure 14-5: The console window

```
CONSOLE: hilbert.pas

Rank of Hilbert matrix (1-5)? 5

Hilbert matrix:

      1.000000        0.500000        0.333333        0.250000        0.200000
      0.500000        0.333333        0.250000        0.200000        0.166667
      0.333333        0.250000        0.200000        0.166667        0.142857
      0.250000        0.200000        0.166667        0.142857        0.125000
      0.200000        0.166667        0.142857        0.125000        0.111111

Hilbert matrix inverted:

     25.002403      -300.051025      1050.241211     -1400.365234       630.180664
   -300.038788      4800.828125    -18903.800781     26885.916016    -12602.961914
   1050.148560    -18903.189453     79394.601563   -117622.859375     56711.464844
  -1400.204224     26884.412109   -117620.234375    179231.750000    -88215.937500
    630.092590    -12602.012695     56709.253906    -88214.539063     44107.308594

Inverse matrix inverted:

      0.999233        0.499423        0.332873        0.249617        0.199672
      0.499685        0.333086        0.249797        0.199828        0.166517
      0.333109        0.249820        0.199850        0.166538        0.142745
      0.249813        0.199848        0.166540        0.142748        0.124904
      0.199835        0.166533        0.142745        0.124903        0.111026

Runtime input text:                                                    Enter
```

The scrollable output area displays the source program's printed runtime output. When the program expects runtime input text, type it into the Runtime input text field and press the Enter button.

Program 14: Pascal IDE

Start the Pascal IDE with a command line similar to

```
java -classpath classes PascalIDE
```

The IDE Process and the Debugger Process
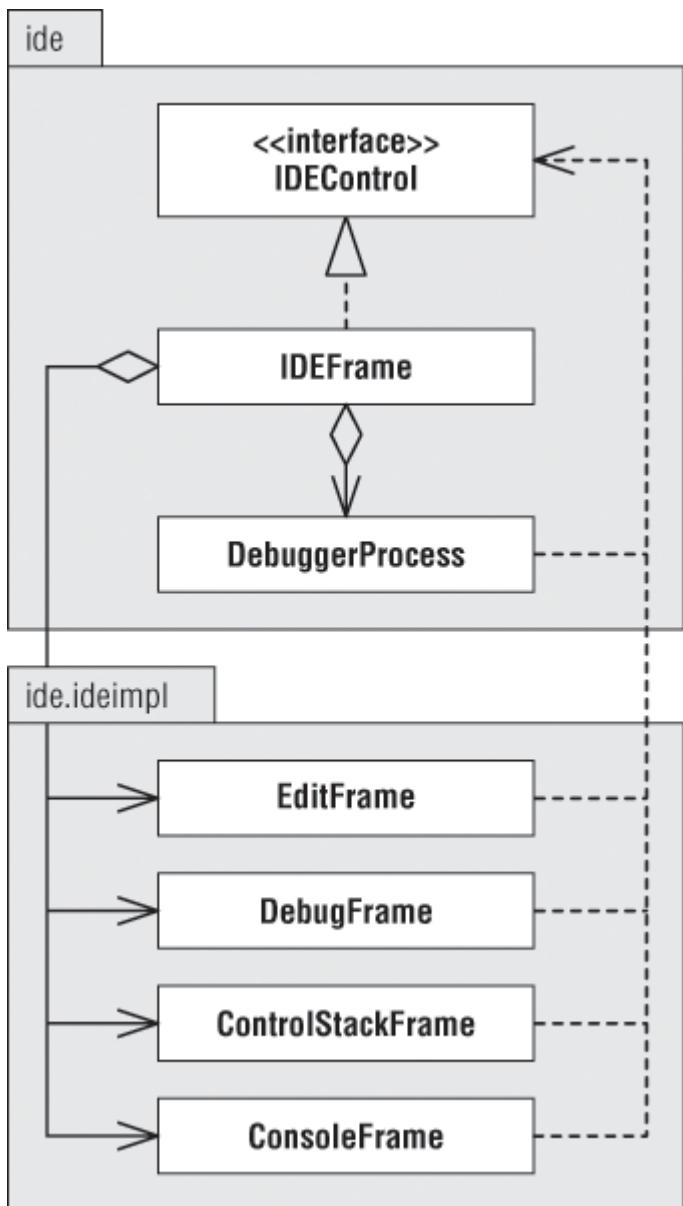
How do you "wrap a GUI" around the interactive source-level debugger that you developed in the previous chapter? Recall how that debugger worked with the command line:

- You typed and entered commands that the debugger parsed and operated upon: single-stepping the execution, setting and unsetting breakpoints, displaying the call stack, etc.
- The debugger wrote short text messages that indicated status information as the interpreter executed the source program, such as the current source line number, the fetched value of a variable, and a new value being assigned to a variable.
- The executing source program read input data and wrote its output as usual.

The IDE Framework

Figure 14-6 shows the UML class diagrams of the Pascal IDE.

Figure 14-6: The Pascal IDE classes
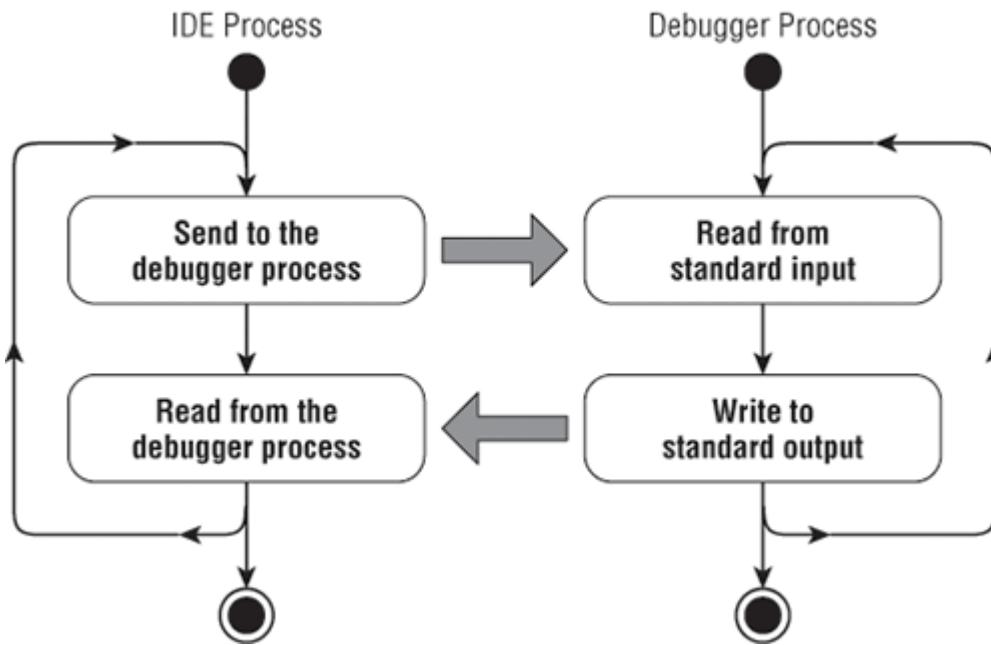
In package `ide`, class `IDEFrame` implements the main IDE window. It owns the internal windows implemented by classes `EditFrame`, `DebugFrame`, `ControlStackFrame`, and `ConsoleFrame` in package `ide.ideimpl`. Class `IDEFrame` also owns the debugger process implemented by class `DebuggerProcess`. Interface `IDEControl` ties everything together: Class `IDEFrame` implements the interface. All the internal window classes and class `DebuggerProcess` depend on the interface.

The IDE framework maintains the current debugger framework and requires only minimal changes to the existing debugger code. If this is done correctly, the debugger does not even realize that the IDE exists. The debugger continues to operate as if you were interacting with it directly via the command line. Accomplish this feat by creating two processes, the IDE process and the debugger process.

Interprocess Communication

Figure 14-7 shows, at the conceptual level, how the IDE process and the debugger process communicate with each other.

Figure 14-7: The IDE and debugger processes

Interprocess communication consists of the debugger's standard input and standard output. The IDE process sends commands and runtime input data to the debugger process, which the debugger reads via its standard input. The IDE process reads status information or program output from the debugger process by reading whatever the debugger writes to its standard output. Each process repeatedly reads and writes until the debugger finishes executing the source program. Whenever a process tries to read and there is no input, the process waits until new input arrives. You've already seen this in the previous chapter: after the command-line debugger prompted you for a debugger command, it waited for you to type and enter the next command.

Therefore, the IDE takes the place of the command-line console, but the debugger still believes it is interacting with the console. When the IDE sends the debugger a command or input data for the executing source program, the debugger thinks it is receiving the text from an incredibly fast typist. When the debugger writes status information or program output, it thinks it is writing to the console, not to the IDE.

Design Note

The key feature of this design is that all of the code you've developed so far in this book, up through the previous chapter, runs in the debugger process. The new code that you will develop in this chapter for the GUI-based IDE runs in the separate IDE process.

Tagged Output

The IDE process must decide what to do with output text it reads from the debugger process. Should it send the text to the debug window's listing or to the console window's output area? Or is the text a status message indicating the line number of the current source line for the debug window to highlight? To simplify making this decision, the debugger writes tagged output. It prefixes with a tag any output line other than runtime output from the source program:

- A source program listing line
- A syntax error message
- A parser summary line
- A debugger status message
- An interpreter summary line

Listing 14-1 shows the tags in interface `IDEControl`. You'll see more of the interface later.

Listing 14-1: Debugger output line tags in interface `IDEControl`

```
// Debugger output line tags.
public static final String LISTING_TAG = "!LISTING:";
public static final String PARSER_TAG = "!PARSER:";
public static final String SYNTAX_TAG = "!SYNTAX:";
public static final String INTERPRETER_TAG = "!INTERPRETER:";

public static final String DEBUGGER_AT_TAG = "!DEBUGGER.AT:";
public static final String DEBUGGER_BREAK_TAG = "!DEBUGGER.BREAK:";
public static final String DEBUGGER_ROUTINE_TAG = "!DEBUGGER.ROUTINE:";
public static final String DEBUGGER_VARIABLE_TAG = "!DEBUGGER.VARIABLE:";
```
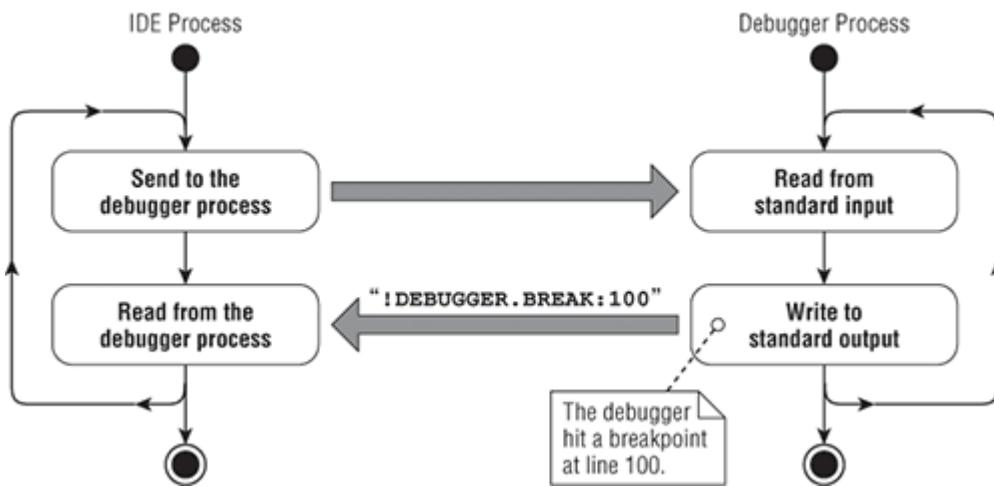
Interprocess Communication Examples

Suppose the debugger is executing a Pascal source program and it hits a breakpoint at source line 100, say. The debugger writes the tagged line

```
!DEBUGGER.BREAK:100
```

to its standard output. See Figure 14-8a.

Figure 14-8a: The debugger hits a breakpoint at source line 100 and writes a tagged output line.

The IDE process, which was waiting for input, reads the tagged line from the debugger process's standard output. The !DEBUGGER.BREAK: tag followed by the 100 tells the IDE process that the debug window should highlight source line 100 in its scrollable listing area.

The debugger process waits for its next command. At this moment, all activity in the IDE has stopped with source line 100 highlighted in the debug window.
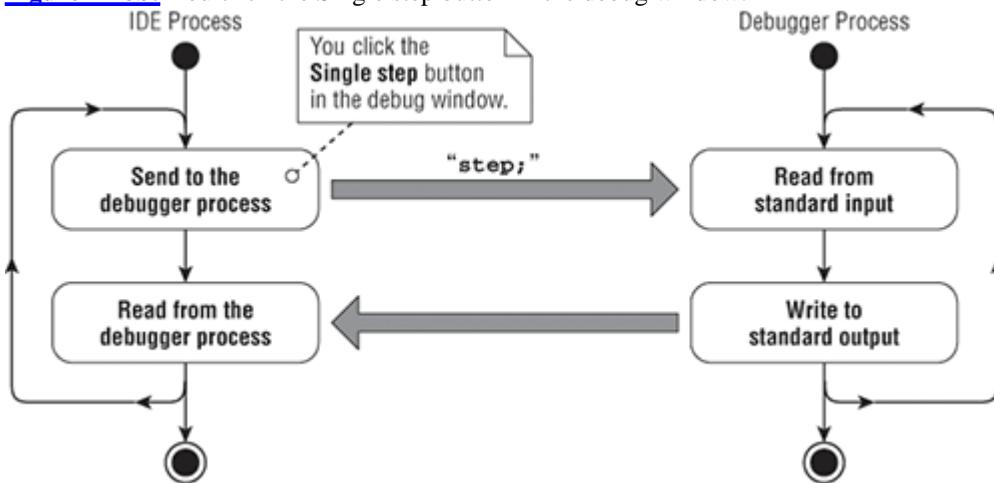
Now suppose you want to single-step to the next source statement. You click the Single step button in the debug window. That action causes the IDE process to write the debugger command

```
step;
```

to the debugger process's standard input.[1] See Figure 14-8b. The debugger process, which was waiting for the next command, reads the step command and proceeds to execute the current source line and then stop at the next source line.

[1] Clicking the Single step button also causes the IDE process to write the stack debugger command to make the debugger process output information about the runtime stack. The IDE process reads this information for the call stack window to update its call stack tree display.

Figure 14-8b: You click the Single step button in the debug window.
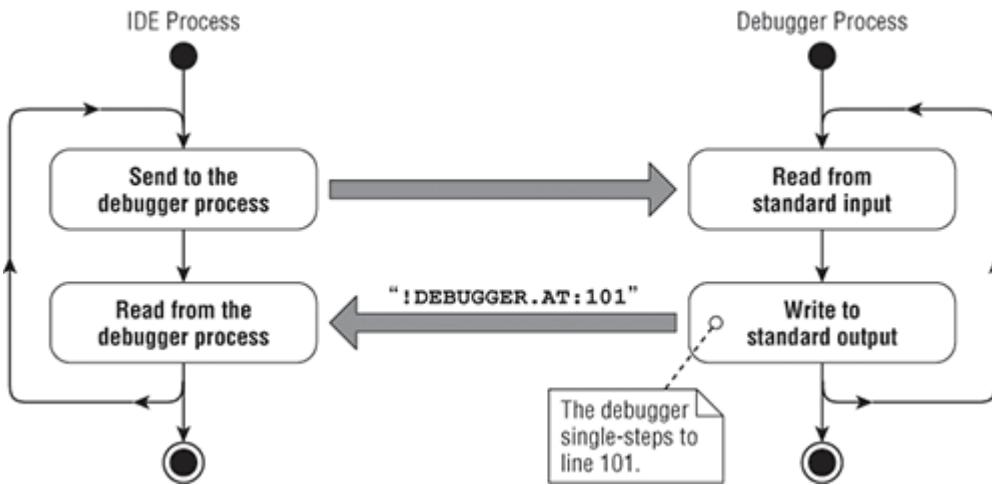
Assuming the next source line number is 101, method atStatement() of class GUIDebugger writes the tagged line

```
!DEBUGGER.AT:101
```

to standard output. See Figure 14-8c. The IDE process reads this tagged line, and the cycle continues.

You'll next see listings of some of the code that accomplishes these actions.

Figure 14-8c: The debugger single-steps to source line 101.

The debugger
single-steps to
line 101.

Writing Tagged Output

Modify the main class `Pascal` to tag parser and interpreter output lines, as shown in <u>Listing 14-2</u>.

<u>Listing 14-2:</u> Tagging output lines in the main class `Pascal`

```
import static wci.ide.IDEControl.*;

    ...

    private static final String PARSER_SUMMARY_FORMAT =
        PARSER_TAG + "%,d source lines, %,d syntax errors, " +
        "%,.2f seconds total parsing time.\n";

    private static final int PREFIX_WIDTH = 5;

    /**
     * Listener for parser messages.
     */
    private class ParserMessageListener implements MessageListener
    {
        /**
         * Called by the parser whenever it produces a message.
         * @param message the message.
         */
        public void messageReceived(Message message)
        {
            MessageType type = message.getType();

            switch (type) {

                case PARSER_SUMMARY: {
                    Number body[] = (Number[]) message.getBody();
                    int statementCount = (Integer) body[0];
                    int syntaxErrors = (Integer) body[1];
                    float elapsedTime = (Float) body[2];

                    System.out.printf(PARSER_SUMMARY_FORMAT,
                                      statementCount, syntaxErrors,
                                      elapsedTime);
                    break;
                }

                case SYNTAX_ERROR: {
                    Object body[] = (Object []) message.getBody();
                    int lineNumber = (Integer) body[0];
                    int position = (Integer) body[1];
                    String tokenText = (String) body[2];
                    String errorMessage = (String) body[3];

                    StringBuilder flagBuffer = new StringBuilder();

                    flagBuffer.append(String.format(SYNTAX_TAG + "%d: %s",
                                                    lineNumber, errorMessage));

                    // Text, if any, of the bad token.
                    if (tokenText != null) {
                        flagBuffer.append(" [at \"").append(tokenText)
                                .append("\"]");
                    }

                    System.out.println(flagBuffer.toString());
                    break;
                }
            }
        }
    }

    private static final String INTERPRETER_SUMMARY_FORMAT =
        INTERPRETER_TAG + "%,d statements executed, %,d runtime errors, " +
        "%,.2f seconds total execution time.\n";

    /**
     * Listener for back end messages.
     */
    private class BackendMessageListener implements MessageListener
    {
        /**
         * Called by the back end whenever it produces a message.
         * @param message the message.
         */
        public void messageReceived(Message message)
        {
            MessageType type = message.getType();

            switch (type) {

                case INTERPRETER_SUMMARY: {
                    Number body[] = (Number[]) message.getBody();
```

```
                    int executionCount = (Integer) body[0];
                    int runtimeErrors = (Integer) body[1];
                    float elapsedTime = (Float) body[2];

                    System.out.printf(INTERPRETER_SUMMARY_FORMAT,
                                      executionCount, runtimeErrors,
                                      elapsedTime);
                    break;
                }

                ...
            }
        }
    }
}
```

The GUI Debugger

The GUI-based IDE uses nearly all of the code you developed in the previous chapter for the command-line debugger. Replace the command-line debugger class `CommandLineDebugger` (review Figure 13-2 and Listing 13-9) with class `GUIDebugger` in package `backend.interpreter.debuggerimpl`. See Listing 14-3.

Listing 14-3: Class `GUIDebugger`

```
/**
 * <h1>GUIDebugger</h1>
 *
 * <p>GUI version of the interactive source-level debugger.</p>
 */
public class GUIDebugger extends Debugger
{
    private CommandProcessor commandProcessor;

    /**
     * Constructor.
     * @param backend the back end.
     * @param runtimeStack the runtime stack.
     */
    public GUIDebugger(Backend backend, RuntimeStack runtimeStack)
    {
        super(backend, runtimeStack);
        commandProcessor = new CommandProcessor(this);
    }

    /**
     * Process a message from the back end.
     * @param message the message.
     */
    public void processMessage(Message message)
    {
        commandProcessor.processMessage(message);
    }

    /**
     * Display a prompt for a debugger command.
     */
    public void promptForCommand() {}

    /**
     * Parse a debugger command.
     * @return true to parse another command immediately, else false.
     */
    public boolean parseCommand()
    {
        return commandProcessor.parseCommand();
    }

    /**
     * Process a source statement.
     * @param lineNumber the statement line number.
     */
    public void atStatement(Integer lineNumber)
    {
        System.out.println(DEBUGGER_AT_TAG + lineNumber);
    }

    /**
     * Process a breakpoint at a statement.
     * @param lineNumber the statement line number.
     */
    public void atBreakpoint(Integer lineNumber)
    {
        System.out.println(DEBUGGER_BREAK_TAG + lineNumber);
    }

    /**
     * Process the current value of a watchpoint variable.
     * @param lineNumber the current statement line number.
     * @param name the variable name.
     * @param value the variable&apos;s value.
     */
    public void atWatchpointValue(Integer lineNumber,
                                  String name, Object value)
    {
    }

    /**
     * Process assigning a new value to a watchpoint variable.
     * @param lineNumber the current statement line number.
     * @param name the variable name.
     * @param value the new value.
     */
    public void atWatchpointAssignment(Integer lineNumber,
                                       String name, Object value)
    {
    }

    /**
     * Process calling a declared procedure or function.
     * @param lineNumber the current statement line number.
     * @param name the routine name.
     */
    public void callRoutine(Integer lineNumber, String routineName)
    {
    }

    /**
```

```
 * Process returning from a declared procedure or function.
 * @param lineNumber the current statement line number.
 * @param name the routine name.
 */
public void returnRoutine(Integer lineNumber, String routineName)
{
}

/**
 * Display a value.
 * @param valueString the value string.
 */
public void displayValue(String valueString)
{
    System.out.println(valueString);
}

/**
 * Display the call stack.
 * @param stack the list of elements of the call stack.
 */
public void displayCallStack(ArrayList stack)
{
    // Call stack header.
    System.out.println(DEBUGGER_ROUTINE_TAG + -1);

    for (Object item : stack) {

        // Name of a procedure or function.
        if (item instanceof SymTabEntry) {
            SymTabEntry routineId = (SymTabEntry) item;
            String routineName = routineId.getName();
            int level = routineId.getSymTab().getNestingLevel();
            Definition definition = routineId.getDefinition();

            System.out.println(DEBUGGER_ROUTINE_TAG + level + ":" +
                                definition.getText().toUpperCase() + " " +
                                routineName);
        }

        // Variable name-value pair.
        else if (item instanceof NameValuePair) {
            NameValuePair pair = (NameValuePair) item;
            System.out.print(DEBUGGER_VARIABLE_TAG +
                             pair.getVariableName() + ":");
            displayValue(pair.getValueString());
        }
    }

    // Call stack footer.
    System.out.println(DEBUGGER_ROUTINE_TAG + -2);
}

/**
 * Terminate execution of the source program.
 */
public void quit()
{
    System.out.println("!INTERPRETER:Program terminated.");
    System.exit(-1);
}

/**
 * Handle a debugger command error.
 * @param errorMessage the error message.
 */
public void commandError(String errorMessage)
{
    runtimeError(errorMessage, 0);
}

/**
 * Handle a source program runtime error.
 * @param errorMessage the error message.
 * @param lineNumber the source line number where the error occurred.
 */
public void runtimeError(String errorMessage, Integer lineNumber)
{
    System.out.print("**** RUNTIME ERROR");
    if (lineNumber != null) {
        System.out.print(" AT LINE " +
                         String.format("%03d", lineNumber));
    }
    System.out.println(": " + errorMessage);
}
}
```
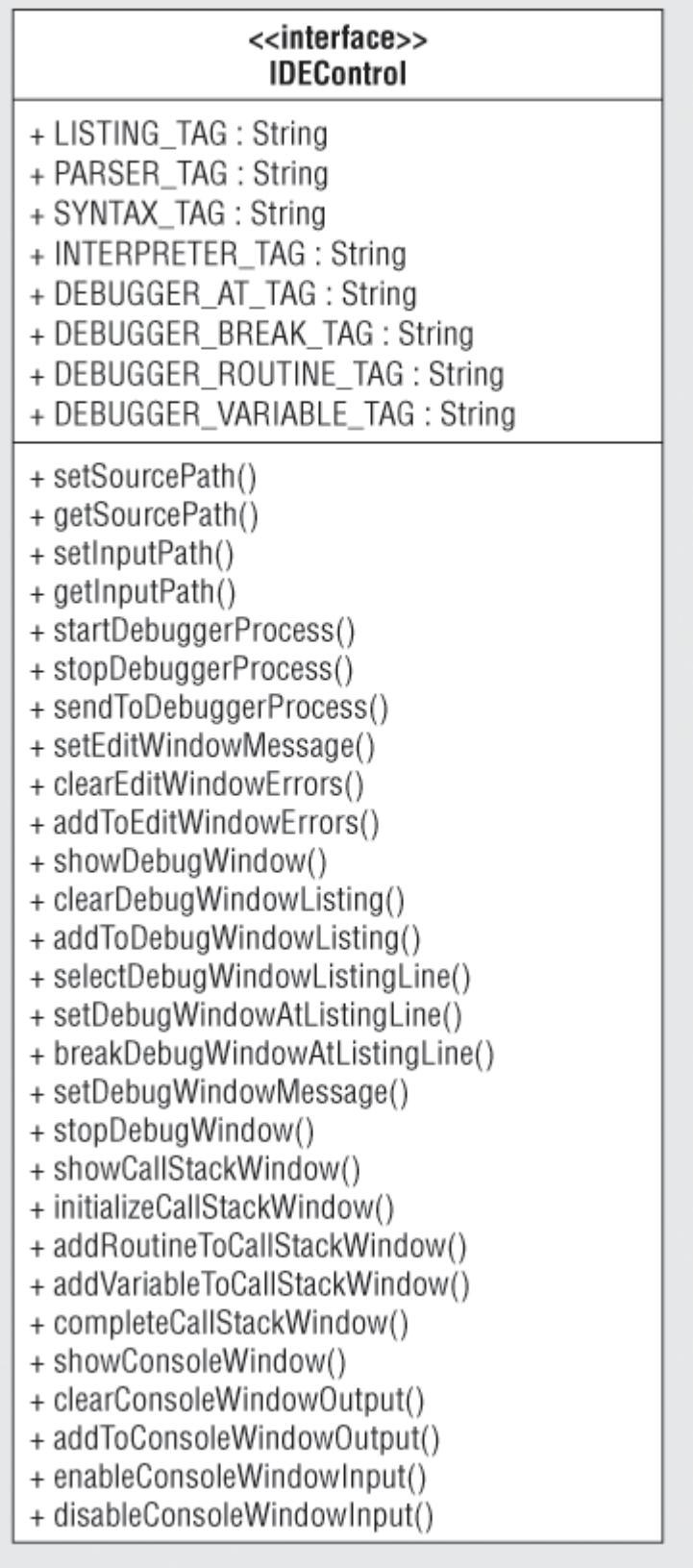
Methods of class `GUIDebugger` are very similar to the corresponding classes of class `CommandLineDebugger`. The key differences are the output line tags, such as those written by method `atStatement()`, and some methods do nothing, such as method `atWatchpointValue()`. Note that method `displayCallStack()` writes a "header" at the start and a "footer" at the end.

The Control Interface

Figure 14-6 showed how the IDE control interface `IDEControl` ties together the components of the IDE framework. Figure 14-9 shows a more detailed UML diagram of this interface. The methods specified by the interface represent the various operations that the IDE process performs in the windows and on the debugger process.

Figure 14-9: Interface `IDEControl`

Listing 14-4 shows the methods of interface `IDEControl`. You've already seen the interface's definitions of the debugger's output line tags.

Listing 14-4: Methods specified by interface `IDEControl`

```
/**
 * <h1>IDEControl</h1>
 *
 * <p>The master interface of the Pascal IDE.</p>
 */
public interface IDEControl
{
    // Debugger output line tags.
    ...

    /**
     * Set the path of the source file.
     * @param sourcePath the path.
```

```java
     */
    public void setSourcePath(String sourcePath);

    /**
     * @return the path of the source file.
     */
    public String getSourcePath();

    /**
     * Set the path of the runtime input data file.
     * @param inputPath the path.
     */
    public void setInputPath(String inputPath);

    /**
     * @return the path of the runtime input data file.
     */
    public String getInputPath();

    /**
     * Start the debugger process.
     * @param sourceName the source file name.
     */
    public void startDebuggerProcess(String sourceName);

    /**
     * Stop the debugger process.
     */
    public void stopDebuggerProcess();

    /**
     * Send a command or runtime input text to the debugger process.
     * @param text the command string or input text.
     */
    public void sendToDebuggerProcess(String text);

    /**
     * Set the editor window&apos;s message.
     * @param message the message.
     * @param color the message color.
     */
    public void setEditWindowMessage(String message, Color color);

    /**
     * Clear the editor window&apos;s syntax errors.
     */
    public void clearEditWindowErrors();

    /**
     * Add a syntax error message to the editor window&apos;s syntax errors.
     * @param line the error message.
     */
    public void addToEditWindowErrors(String line);

    /**
     * Show the debugger window.
     * @param sourceName the source file name.
     */
    public void showDebugWindow(String sourceName);

    /**
     * Clear the debugger window&apos;s listing.
     */
    public void clearDebugWindowListing();

    /**
     * Add a line to the debugger window&apos;s listing.
     * @param line the listing line.
     */
    public void addToDebugWindowListing(String line);

    /**
     * Select a listing line in the debugger window.
     * @param lineNumber the line number.
     */
    public void selectDebugWindowListingLine(int lineNumber);

    /**
     * Set the debugger to a listing line.
     * @param lineNumber the line number.
     */
    public void setDebugWindowAtListingLine(int lineNumber);

    /**
     * Set the debugger to break at a listing line.
     * @param lineNumber the line number.
     */
    public void breakDebugWindowAtListingLine(int lineNumber);

    /**
     * Set the debugger window&apos;s message.
     * @param message the message.
     * @param color the message color.
     */
    public void setDebugWindowMessage(String message, Color color);

    /**
     * Stop the debugger.
     */
    public void stopDebugWindow();

    /**
     * Show the call stack window.
     * @param sourceName the source file name.
     */
    public void showCallStackWindow(String sourceName);

    /**
     * Initialize the call stack display.
     */
    public void initializeCallStackWindow();

    /**
     * Add an invoked routine to the call stack display.
     * @param level the routine&apos;s nesting level.
     * @param header the routine&apos;s header.
     */
    public void addRoutineToCallStackWindow(String level, String header);

    /**
     * Add a local variable to the call stack display.
     * @param name the variable&apos;s name.
```

```
     * @param value the variable&apos;s value.
     */
    public void addVariableToCallStackWindow(String name, String value);

    /**
     * Complete the call stack display.
     */
    public void completeCallStackWindow();

    /**
     * Show the console window.
     * @param sourceName the source file name.
     */
    public void showConsoleWindow(String sourceName);

    /**
     * Clear the console window&apos;s output.
     */
    public void clearConsoleWindowOutput();

    /**
     * Add output text to the console window.
     * @param line the output text.
     */
    public void addToConsoleWindowOutput(String text);

    /**
     * Enable runtime input from the console window.
     */
    public void enableConsoleWindowInput();

    /**
     * Disable runtime input from the console window.
     */
    public void disableConsoleWindowInput();
}
```

Class `IDEFrame` implements all of the methods of interface `IDEControl`. Therefore, whenever one window needs to cause some operation to happen in another window or in the debugger process, it routes the request through class `IDEFrame`. (Recall from Figure 14-6 that class `IDEFrame` owns all the internal window classes and the debugger process class, and the internal window classes and the debugger process classes all depend on interface `IDEControl`.)

For example, Listing 14-5 shows the action method `singleStepAction()` in class `DebugFrame` for the Single step button.

Listing 14-5: Method `singleStepAction()` of class `DebugFrame`

```
/**
 * <h1>DebugFrame</h1>
 *
 * <p>The debug window of the Pascal IDE.</p>
 */
public class DebugFrame
    extends JInternalFrame
    implements ActionListener
{
    ...
    private IDEControl control;
    ...

    /**
     * Constructor.
     */
    public DebugFrame()
    {
        try {
            initGuiComponents();
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Constructor.
     * @param control the IDE control.
     */
    public DebugFrame(IDEControl control)
    {
        this();
        this.control = control;

        ...
    }

    /**
     * Single step button event handler.
     */
    private void singleStepAction()
    {
        ...

        control.sendToDebuggerProcess("step;");
        control.sendToDebuggerProcess("stack;");
        control.enableConsoleWindowInput();
    }

    ...
}
```

Whenever you click the Single step button, method `singleStepAction()` calls `sendToDebuggerProcess()` to send the step and stack commands to the debugger process, and it also calls `enableConsoleWindowInput()` to enable the console window to accept any input. Class `IDEFrame` implements these two methods, as shown in Listing 14-6.

Listing 14-6: Methods `sendToDebuggerProcess()` and `enableConsoleWindowInput()` of class `IDEFrame`

```
/**
 * <h1>IDEFrame</h1>
 *
 * <p>The main window of the Pascal IDE.</p>
```

```
    */
public class IDEFrame
    extends JFrame
    implements IDEControl
{
    ...
    private EditFrame editFrame;
    private DebugFrame debugFrame;
    private ConsoleFrame consoleFrame;
    private CallStackFrame stackFrame;

    private DebuggerProcess debuggerProcess;
    ...

    /**
     * Send a command or runtime input text to the debugger process.
     * @param text the command string or input text.
     */
    public void sendToDebuggerProcess(String text)
    {
        debuggerProcess.writeToDebuggerStandardInput(text);
    }

    ...

    /**
     * Enable runtime input from the console window.
     */
    public void enableConsoleWindowInput()
    {
        consoleFrame.enableInput();
    }

    ...
}
```
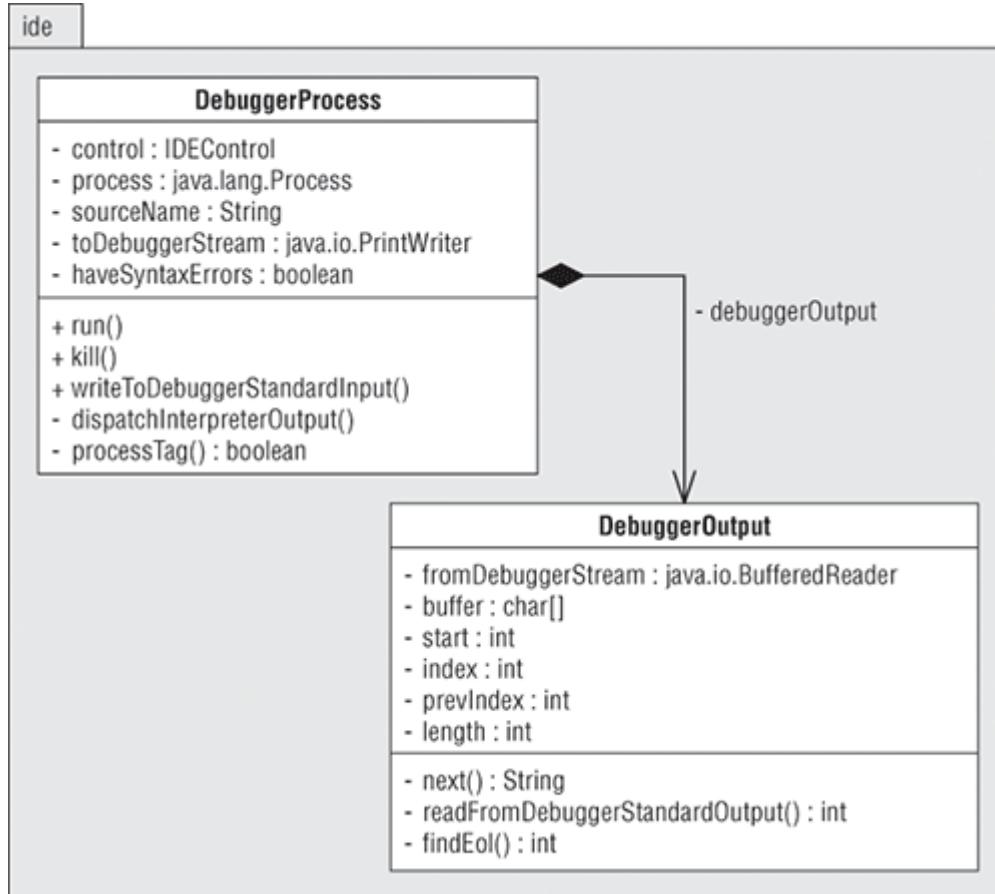
The Debugger Process

Figure 14-6 showed how class `IDEFrame` owns the debugger process. Figure 14-10 shows a more detailed UML diagram of class `DebuggerProcess` and its inner class `DebuggerOutput`.

Figure 14-10: Class `DebuggerProcess` and its inner class `DebuggerOutput`

Class `DebuggerProcess` owns and controls the debugger process, referred to by its private field `process`. Field `toDebuggerStream` is the I/O stream from the IDE process to the debugger process's standard input. The IDE process will write to this `java.io.PrintWriter` stream. Inner class `DebuggerOutput` represents output from the debugger processes. Its field `fromDebuggerStream` is the I/O stream from the debugger process's standard output to the IDE process. The IDE process will read from this `java.io.BufferedReader` stream.

Listing 14-7 shows the fields and public methods of class `DebuggerProcess`.

Listing 14-7: Fields and public methods of class `DebuggerProcess`

```
/**
 * <h1>DebuggerProcess</h1>
 *
```

```
 * <p>The debugger process of the Pascal IDE.</p>
 */
public class DebuggerProcess extends Thread
{
    private IDEControl control;            // the IDE control interface
    private Process process;               // the debugger process
    private String sourceName;             // source file name
    private PrintWriter toDebuggerStream;  // IDE to debugger I/O stream
    private DebuggerOutput debuggerOutput; // debugger process output
    private boolean haveSyntaxErrors = false;  // true if have syntax errors
    private boolean debugging = false;         // true if debugging process I/O

    /**
     * Constructor.
     * @param control the IDE control.
     * @param sourceName the source file name.
     */
    public DebuggerProcess(IDEControl control, String sourceName)
    {
        this.control = control;
        this.sourceName = sourceName;
    }

    // The command that starts the debugger process.
    private static final String COMMAND =
                            "java -classpath classes Pascal execute %s %s";

    /**
     * Run the procecess.
     */
    public void run()
    {
        try {
            // Start the Pascal debugger process.
            String command = String.format(COMMAND, control.getSourcePath(),
                                            control.getInputPath());
            process = Runtime.getRuntime().exec(command);

            // Capture the process&apos;s input stream.
            toDebuggerStream = new PrintWriter(process.getOutputStream());

            // Read and dispatch output text from the
            // debugger process for processing.
            debuggerOutput = new DebuggerOutput(process);
            dispatchDebuggerOutput();
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Kill the debugger process.
     */
    public void kill()
    {
        if (process != null) {
            process.destroy();
            process = null;
        }
    }

    /**
     * Write a command or runtime input text
     * to the debugger process&apos;s standard input.
     * @param text the command string or input text.
     */
    public void writeToDebuggerStandardInput(String text)
    {
        synchronized(DebuggerProcess.class) {
            toDebuggerStream.println(text);
            toDebuggerStream.flush();

            if (debugging) {
                System.err.println("Sent: &apos;" + text + "&apos;");
            }
        }
    }

    ...
}
```

Class `DebuggerProcess` extends `java.lang.Thread`. Its `run()` method starts the debugger process by calling `Runtime.getRuntime ().exec()`. The `new PrintWriter(process.getOutputStream())` creates a `PrintWriter` that writes to the debugger process's standard input. The `new DebuggerOutput(process)` creates an instance of the internal class that reads from the debugger process's standard output. Method `dispatchDebuggerOutput()` processes output from the debugger.

Method `kill()` terminates the debugger process, and method `writeToDebuggerStandardInput()` writes to the debugger process.

Listing 14-8 shows methods `dispatchDebuggerOutput()` and `processTag()`.

Listing 14-8: Methods `dispatchDebuggerOutput()` and `processTag()` of class `DebuggerProcess`

```
    /**
     * Read and dispatch output text from the debugger process for processing.
     * @throws Exception if an error occurred.
     */
    private void dispatchDebuggerOutput()
        throws Exception
    {
        String text;

        // Loop to process debugger output text
        // which may contain embedded output tags.
        do {
            text = debuggerOutput.next();
            if (debugging) {
                System.out.println("Read: &apos;" + text + "&apos;");
            }

            int index;
            do {
                Thread.sleep(1);
                index = text.indexOf(&apos;!&apos;);

                // The debugger output text contains the ! character.
                // It may be the start of an output tag.
                if (index >= 0) {
```

```java
                    // Add any preceding text to the console window.
                    if (index > 0) {
                        String precedingText = text.substring(0, index);
                        control.addToConsoleWindowOutput(precedingText);
                        text = text.substring(index);
                    }

                    // Yes, it was an output tag. Don&apos;t loop again.
                    if (processTag(text)) {
                        index = -1;
                    }

                    // No, it wasn&apos;t.
                    // Loop again to process the rest of the output text.
                    else {
                        control.addToConsoleWindowOutput("!");
                        text = text.substring(1);
                    }
                }

                // Send all the debugger output text to the console window.
                else {
                    control.addToConsoleWindowOutput(text);
                }
            } while (index >= 0);
        }
        while (!text.startsWith(INTERPRETER_TAG));
}

/**
 * Process a tag in the output text.
 * @param text the output text
 * @return true if processed a tag, else false.
 */
private boolean processTag(String text)
{
    // Listing line.
    if (text.startsWith(LISTING_TAG)) {
        control.addToDebugWindowListing(
                    text.substring(LISTING_TAG.length()));
        return true;
    }

    // Syntax error message.
    else if (text.startsWith(SYNTAX_TAG)) {
        String errorMessage = text.substring(SYNTAX_TAG.length());
        control.addToEditWindowErrors(errorMessage);
        haveSyntaxErrors = true;
        return true;
    }

    // Parser message.
    else if (text.startsWith(PARSER_TAG)) {
        control.setEditWindowMessage(text.substring(PARSER_TAG.length()),
                                     haveSyntaxErrors ? Color.RED
                                                      : Color.BLUE);

        if (!haveSyntaxErrors) {
            control.clearEditWindowErrors();
            control.setDebugWindowMessage("", Color.BLUE);
            control.showDebugWindow(sourceName);
            control.showCallStackWindow(sourceName);
            control.showConsoleWindow(sourceName);
        }
        else {
            control.setDebugWindowMessage("Fix syntax errors.",
                                          Color.RED);
            control.stopDebugWindow();
            control.disableConsoleWindowInput();
        }

        return true;
    }

    // Debugger at a source statement.
    else if (text.startsWith(DEBUGGER_AT_TAG)) {
        String lineNumber = text.substring(DEBUGGER_AT_TAG.length());
        control.setDebugWindowAtListingLine(
                    Integer.parseInt(lineNumber.trim()));
        control.setDebugWindowMessage(" ", Color.BLUE);
        return true;
    }

    // Debugger break at a source statement.
    else if (text.startsWith(DEBUGGER_BREAK_TAG)) {
        String lineNumber = text.substring(DEBUGGER_BREAK_TAG.length());
        control.breakDebugWindowAtListingLine(
                    Integer.parseInt(lineNumber.trim()));
        control.setDebugWindowMessage("Break at text " + lineNumber,
                                      Color.BLUE);
        return true;
    }

    // Debugger add a routine to the call stack.
    else if (text.startsWith(DEBUGGER_ROUTINE_TAG)) {
        String components[] = text.split(":");
        String level = components[1].trim();

        // Header.
        if (level.equals("-1")) {
            control.initializeCallStackWindow();
        }

        // Footer.
        else if (level.equals("-2")) {
            control.completeCallStackWindow();
        }

        // Routine name.
        else {
            String header = components[2].trim();
            control.addRoutineToCallStackWindow(level, header);
        }

        return true;
    }

    // Debugger add a local variable to the call stack.
    else if (text.startsWith(DEBUGGER_VARIABLE_TAG)) {
        text = text.substring(DEBUGGER_VARIABLE_TAG.length());
```

```
                int index = text.indexOf(":");
                String name = text.substring(0, index);
                String value = text.substring(index + 1);

                control.addVariableToCallStackWindow(name, value);
                return true;
            }

            // Interpreter message.
            else if (text.startsWith(INTERPRETER_TAG)) {
                control.setDebugWindowMessage(
                    text.substring(INTERPRETER_TAG.length()), Color.BLUE);
                control.stopDebugWindow();
                control.disableConsoleWindowInput();
                return true;
            }
            else {
                return false;  // it wasn't an output tag
            }
        }
    }
```

Methods `dispatchDebuggerOutput()` and `processTag()` together read the output from the debugger process and decide what to do with it. The outer loop of `dispatchDebuggerOutput()` calls `debuggerOutput.next()` to get the next output from the debugger process. The inner loop looks for any output tags. Any output that is not tagged is runtime output from the source program, and the method sends such output to the console window with calls to `control.addToConsoleWindowOutput()`.

Method `processTag()` decides what to do with tagged output. It calls the appropriate `IDEControl` method according to each tag.

shows the inner class `DebuggerOutput`.

** Inner class `DebuggerOutput` of class `DebuggerProcess`**

```
/**
 * Output from the debugger.
 */
private class DebuggerOutput
{
    private static final int BUFFER_SIZE = 1024;

    private BufferedReader fromDebuggerStream; // debugger to IDE I/O stream
    private char buffer[];                     // output buffer
    private int start;                         // start of output line
    private int index;                         // index of \n or end of line
    private int prevIndex;                     // previous index
    private int length;                        // output text length

    /**
     * Constructor.
     * @param process the interpreter process.
     */
    private DebuggerOutput(Process process)
    {
        fromDebuggerStream = new BufferedReader(
                                  new InputStreamReader(
                                       process.getInputStream()));
        buffer = new char[BUFFER_SIZE];
        start = 0;
        length = 0;
    }

    /**
     * Get the next complete or partial output line.
     * @return the output line.
     * @throws Exception if an error occurred.
     */
    private String next()
        throws Exception
    {
        String output = "";

        // Loop to process output from the interpreter.
        for (;;) {
            Thread.sleep(1);
            index = findEol(prevIndex);

            // Found end of line: Return the line.
            if (index < length) {
                output += new String(buffer, start, index - start + 1);
                start = index + 1;
                prevIndex = start;

                if (debugging) {
                    System.err.println("Output: '" + output + "'");
                }

                return output;
            }

            // No end of line: Append to the current output.
            if (index > start) {
                output += new String(buffer, start, index - start);
            }

            // Prepare to read again into the buffer.
            start = 0;
            length = 0;
            prevIndex = 0;

            // Read more output if available.
            if (fromDebuggerStream.ready()) {
                length = readFromDebuggerStandardOutput();
            }

            // No more output: Return the current output.
            else {
                if (debugging) {
                    System.err.println("Output: '" + output + "'");
                }

                return output;
            }
        }
    }

    /**
     * Read debugger status or runtime output
     * from the debugger's standard output.
     * @return the number of characters read.
```

```
     * @throws Exception if an error occurred.
     */
    private int readFromDebuggerStandardOutput()
        throws Exception
    {
        return fromDebuggerStream.read(buffer);
    }

    /**
     * Look for \n in the output.
     * @param index where to start looking.
     * @return the index of \n or the end of output.
     */
    private int findEol(int index)
    {
        while ((index < length) && (buffer[index] != '\n')) {
            ++index;
        }

        return index;
    }
}
```

Field `fromDebuggerStream` is a `BufferedReader` that reads from the debugger process's standard output and puts the characters into `buffer`. The integer fields `start`, `index`, `prevIndex`, and `length` keep track of locations in this input buffer.

Method `next()` returns complete lines (each terminated by the end-of-line character `'\n'`). It loops to read output characters from the debugger process until it has a complete line to return. However, if it only has a partial line and there is no more output from the debugger process, the method will return a partial line. This can happen, for example, if the Pascal source program executed a call to the standard procedure `write` rather than `writeln`.

Method `readFromDebuggerStandardOutput()` does the reading from the debugger process's standard output. Method `findEol()` looks for end-of-line characters in the debugger output.

This concludes this part of the book about developing an interpreter back end and an interactive debugger. Once you had a working Pascal interpreter (Chapter 12), it was relatively straightforward to add an interactive source-level debugger (Chapter 13) and then build an IDE that wraps a GUI around the debugger (this chapter).

The next chapter is the first of several about developing a compiler back end that generates object code for the Java Virtual Machine (JVM).

Chapter 15

Jasmin Assembly Language and Code Generation for the Java Virtual Machine

You now shift gears and move from a Pascal interpreter to building a Pascal compiler. This chapter lays important groundwork for code generation in the compiler back end. It introduces the architecture of the Java Virtual Machine (JVM) and the Jasmin assembly language.

Goals and Approach

As first described in Chapter 1, a compiler generates object code to be executed by a particular machine. This can be an actual machine or a virtual machine. The object code can be in machine language or in an assembly language. In this book, the compiler will generate object code in the form of assembly language for the Java Virtual Machine.

This chapter describes the architecture of the JVM. It won't be a complete description, just enough for the Pascal compiler to generate object code for Pascal source programs in the next three chapters. This chapter will introduce just enough of the Jasmin assembly language that was tailored for the JVM.

You'll also examine some of the code generation classes in the compiler back end that you'll use in the following chapters. The compiler will reuse, with only a few modifications, the front end and middle tier components that you've already developed.

Organization of the Java Virtual Machine
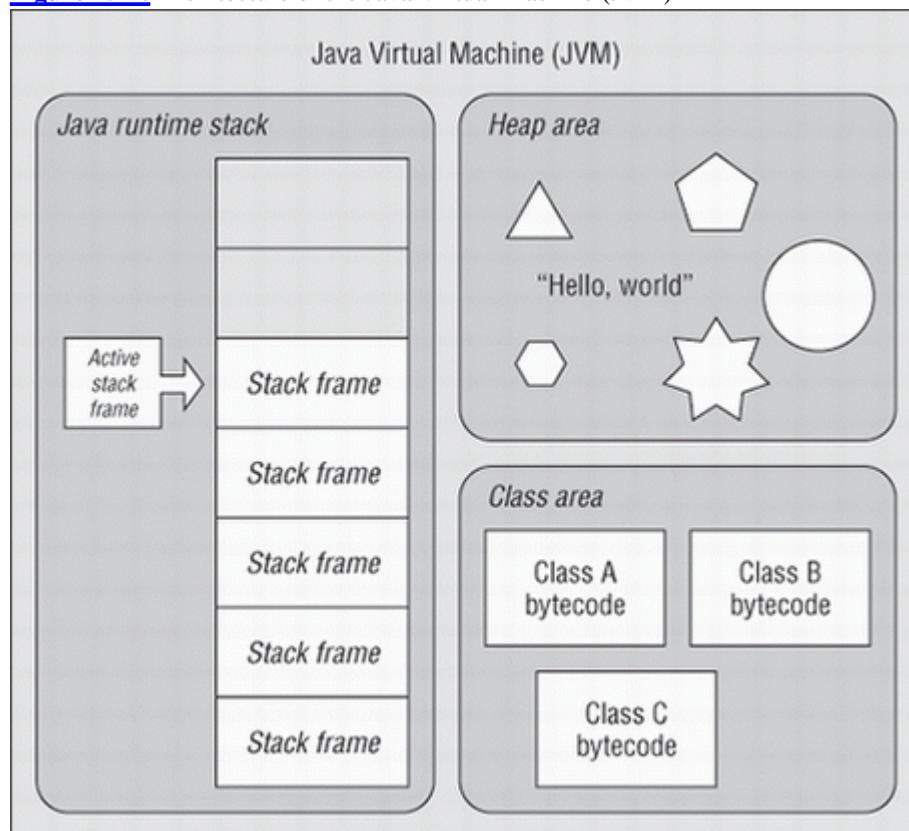
The Java Virtual Machine, or JVM, is usually implemented in software (i.e., it's a program) rather than by the hardware of chips and other electronic components; hence, it's *virtual*. Since it's a virtual machine, it can be developed to run on many actual machines, which greatly enhances the portability of Java across hardware platforms.

The JVM is a stack-oriented architecture that employs a runtime stack. You implemented such an architecture for the Pascal interpreter in the previous chapters.

The JVM is organized into four major areas: the class area, the Java runtime stack, the heap, and the native method stacks. The compiler you'll develop in subsequent chapters will work only with the first three areas. See
Figure 15-1.

Figure 15-1: Architecture of the Java Virtual Machine (JVM)

The Class Area

The class area holds all of an application's classes and its constants. At run time, the Java class loader loads needed classes into the part of the class area called the *method area*. Constants are loaded into the *constant pool* part of the class area.

Because the Java class loader does such a good job of managing the class area at run time, you generally do not need to be concerned with this area.

The Heap Area

Java runtime objects that are dynamically allocated by a method (with the `new` operator) are kept in the runtime heap area. Therefore, each object reference in a local variables array points to the object in the heap.

The JVM's heap manager allocates objects that are created by a method's instructions. It employs a garbage collection algorithm to deallocate objects that are no longer referenced.

Add heap management to the list of runtime tasks that the JVM takes care of extremely well and that you don't have to worry about.

The Java Runtime Stack

The Pascal interpreter pushed an activation record onto the runtime stack during a call to a declared procedure or function, and it popped off the activation record upon the return from the routine. The JVM has a similar concept, which it calls a *stack frame*. A stack frame represents the invocation of a method, and the topmost frame, the *active stack frame*, represents the currently executing method. In the next chapter, you'll compile each Pascal procedure or function into a method.

Each stack frame contains a *program counter* that keeps track of the currently executing instruction. When a method is called, the JVM pushes a new stack frame onto the runtime stack, and this frame will contain its own program counter to keep track of the executing instruction in the called routine. When the method returns, the JVM pops the method's stack frame off the stack, the previous stack frame becomes the active one again, and its program counter allows execution to resume with the instruction following the call. The JVM does an excellent job managing the program counters, and so you needn't be too concerned with them.
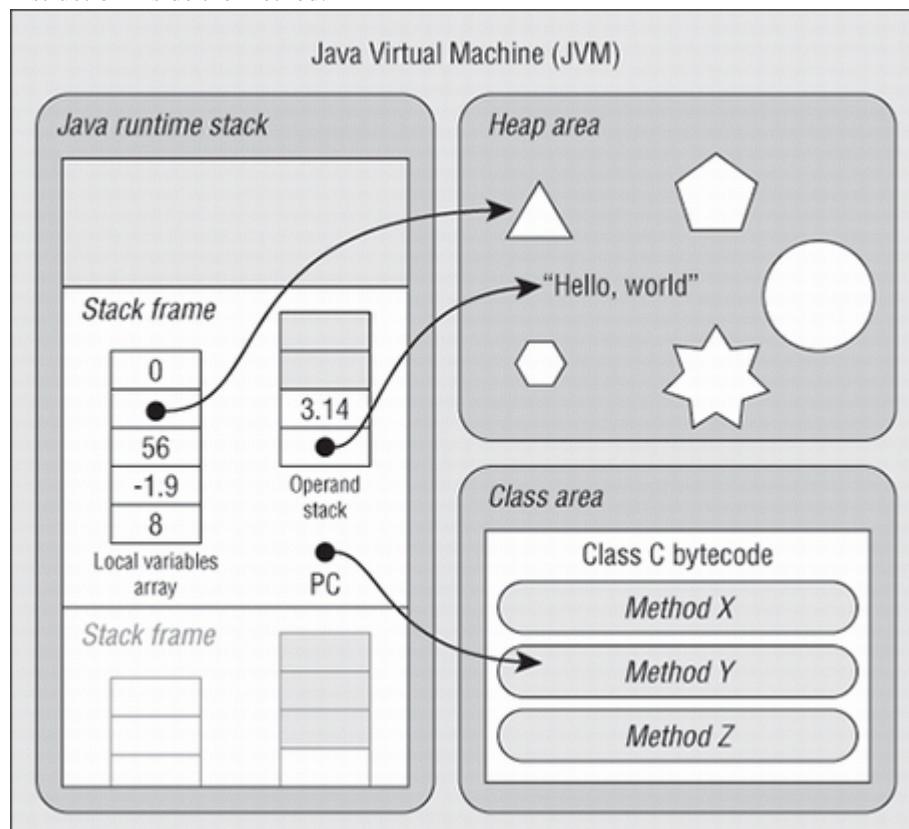
A stack frame also contains an *operand stack* and a *local variables array*. See Figure 15-2.

The Operand Stack

Each stack frame contains an operand stack. A method's code uses the operand stack to perform its arithmetic, logical, and control operations by pushing and popping values. Each element of the operand stack can contain an integer or floating-point scalar value, or a reference to (i.e., the address of) a dynamically allocated object or string in the heap. Boolean and character scalar values are stored as integers.

The size of the operand stack (the maximum number of elements that can be pushed onto it) in a stack frame depends on the Jasmin method that was called, and so operand stacks will have different sizes in different stack frames.

Figure 15-2: Contents of the stack frame while executing method Y of class `c`. The program counter (PC) points to the current instruction inside the method.

The Local Variables Array

Each method can have formal parameters and local variables. While a method is executing, the values of its parameters and variables are maintained in the stack frame's local variables array. Each parameter and variable is assigned a slot, or element, of the array to store its value. A value can be a scalar value or a reference to a dynamically allocated object or string in the heap.

The compiler may allocate additional slots in the array for temporary variables. You can think of the local variables array as a set of registers. Each Jasmin method invocation gets its own set of registers when the JVM pushes method's stack frame onto the runtime stack.

The number of slots contained by the local variables array in a stack frame depends on the Jasmin method that was called, and so local variables arrays will have different sizes in different stack frames.

JVM Limitations

As you might expect, the JVM is very well-suited for the Java programming language. It also does not contain many features that are not required by Java.

One major difference between the Pascal language and the Java language is nested procedures and functions. As you've seen, Pascal programs can have procedures and functions declared within each other to arbitrary depths. As described in Chapter 12, the Pascal interpreter handled nested scopes by using a runtime display to access values on the runtime stack.

Java, on the other hand, does not allow a method to have nested methods. The JVM does not have a runtime display mechanism. You may be able to "fake" nested procedures and functions with inner classes, but that would be rather awkward. Therefore, to simplify the compiler development in the next chapters, you'll simply adopt the rule: *No Pascal source programs with nested procedures or functions.* You'll modify the front end parser to allow only procedures and functions declared at nesting level 1.

The Jasmin Assembly Language

Jasmin is the assembly language for the Java Virtual Machine. The compiler will generate Jasmin code to run on the JVM rather than generate machine language code directly. (JVM machine language code is also known as "bytecode".) Once the compiler has compiled a Pascal source program into an equivalent Jasmin assembly language program, you must run the Jasmin assembler[1] to translate the assembly language program into bytecode. Just like for a Java class, the bytecode for a Jasmin class is stored in a *class file* whose name ends with `.class`. Then you can execute the byte code program on the JVM. (Review Chapter 1, especially the diagram on the right in Figure 1-1.)

[1] Download the Jasmin assembler from
http://jasmin.sourceforge.net/.

Assembly Statements

A Jasmin assembly language program consists of statements, one per line. A statement can be an instruction (and any operands) or a directive (and any operands). There can also be a statement label on a separate line.

Code Generation Methods

Class `CodeGenerator` has been in the `backend.compiler` package since Chapter 2. It becomes the base class of all the code generation subclasses that you'll develop in the next three chapters. This is a good place to put code generation methods that emit parts of Jasmin statements, since all the code generation subclasses will need them.

Listing 15-1 shows the constructors and the `process()` and `generate()` methods.

Listing 15-1: The beginning of class `CodeGenerator`

```
/**
 * <h1>CodeGenerator</h1>
 *
 * <p>The code generator for a compiler back end.</p>
 */
public class CodeGenerator extends Backend
{
    private static PrintWriter assemblyFile;
    private static int instructionCount = 0;

    protected static String programName;

    protected LocalVariables localVariables;
    protected LocalStack localStack;

    /**
     * Constructor.
     */
    public CodeGenerator() {}

    /**
     * Constructor for subclasses.
     * @param the parent code generator.
     */
    public CodeGenerator(CodeGenerator parent)
    {
        super();
        this.localVariables = parent.localVariables;
        this.localStack = parent.localStack;
    }

    /**
     * Process the intermediate code and the symbol table generated by the
     * parser to generate machine-language instructions.
     * @param iCode the intermediate code.
```

```
    * @param symTabStack the symbol table stack.
    * @throws Exception if an error occurred.
    */
   public void process(ICode iCode, SymTabStack symTabStack)
       throws Exception
   {
       this.symTabStack = symTabStack;
       long startTime = System.currentTimeMillis();

       SymTabEntry programId = symTabStack.getProgramId();
       programName = programId.getName();
       String assemblyFileName = programName + ".j";

       // Open a new assembly file for writing.
       assemblyFile = new PrintWriter(
                          new PrintStream(
                              new File(assemblyFileName)));

       // Generate code for the main program.
       CodeGenerator programGenerator = new ProgramGenerator(this);
       programGenerator.generate(iCode.getRoot());
       assemblyFile.close();

       // Send the compiler summary message.
       float elapsedTime = (System.currentTimeMillis() - startTime)/1000f;
       sendMessage(new Message(COMPILER_SUMMARY,
                               new Number[] {instructionCount,
                                             elapsedTime}));
   }

   /**
    * Generate code for a statement.
    * To be overridden by the code generator subclasses.
    * @param node the root node of the statement.
    */
   public void generate(ICodeNode node)
       throws PascalCompilerException
   {
   }

   /**
    * Generate code for a routine.
    * To be overridden by the code generator subclasses.
    * @param routineId the routine&apos;s symbol table entry.
    */
   public void generate(SymTabEntry routineId)
       throws PascalCompilerException
   {
   }
   ...
}
```

Method `process()` sets the static field `programName` and creates an output file to contain the generated assembly object code. It has the same base name as the Pascal source file and has the `.j` suffix, which is the convention for Jasmin files. It calls `programGenerator.generate()`, which you'll develop in the next chapter.

The code generation subclasses of `CodeGenerator` will override the two `generate()` methods.

You'll examine classes `LocalVariables` and `LocalStack` shortly. Their methods keep track of a Jasmin method's local variables array and operand stack, respectively. As you'll see in the next chapter, the program generator subclass `ProgramGenerator` and declared routine generator subclass `DeclaredRoutineGenerator` will each create new `LocalVariables` and `LocalStack` objects, and then the objects will be passed via the subclass constructor to other code generators.

Instructions

A Jasmin instruction statement consists of an operation *mnemonic* (a reserved Jasmin word that represents a single operation code), or *opcode*, in the class file. Many instructions have no operands (any needed values are at the top of the operand stack), and other instruction statements include one, two, or three explicit operands after the operation mnemonic.

An example of an instruction that has no *explicit* operands but requires two *implicit* operand values on the operand stack:

```
IADD
```

The integer add instruction pops off two integer values at the top of the operand stack, adds them, and pushes the integer sum onto the operand stack.

The following store instruction pops off the floating-point value at the top of the stack and stores it into the local variable whose value is in slot 5 of the local variables array. It requires the explicit operand 5.

```
FSTORE 5
```

The instruction

```
getstatic Newton/number I
```

has two explicit operands separated by a blank. It pushes the integer value of the static field `number` of class `Newton` onto the operand stack. It does not require any operand values on the operand stack. The second explicit operand, `I`, is the *type descriptor* for a scalar integer.

The last example above illustrates a couple of Jasmin features. The operand mnemonic is not case sensitive, so you can use `GETSTATIC` or `getstatic`. Whereas Java uses the period `.` to separate components of a name, as in `newton.number`, Jasmin uses the slash `/`.

Directives

Directive statements provide information to the Jasmin assembler that may affect the contents of the class file. A directive always begins with a period immediately followed by the reserved directive word. For example, the `.method` directive starts a Jasmin method, and the `.end` directive ends it. Directive statements can also include various keywords and operands.

For example, the main method of a Java program always has the form

```
public static void main(String args[])
```

The equivalent Jasmin directive is

```
.method public static main([Ljava/lang/String;)V
```

where `public` and `static` are keywords and `main([Ljava/lang/String;)V` is the *signature* of the main method and its return type. A method's signature consists of the method's name and an encoding of its formal parameter types. As explained further below, `[Ljava/lang/String;` is the type descriptor for "array of `String`". The `V` at the end means the method returns `void` (nothing).

shows the enumerated type `Directive` that contains all the directives (with keywords) that the Pascal compiler generates.

Listing 15-2: Enumerated type `Directive`

```
package wci.backend.compiler;

/**
 * <h1>Directive</h1>
 *
 * <p>Jasmin directives.</p>
 */
public enum Directive
{
    CLASS_PUBLIC(".class public"),
    END_CLASS(".end class"),
    SUPER(".super"),
    FIELD_PRIVATE_STATIC(".field private static"),
    METHOD_PUBLIC(".method public"),
    METHOD_STATIC(".method static"),
    METHOD_PUBLIC_STATIC(".method public static"),
    METHOD_PRIVATE_STATIC(".method private static"),
    END_METHOD(".end method"),
    LIMIT_LOCALS(".limit locals"),
    LIMIT_STACK(".limit stack"),
    VAR(".var"),
    LINE(".line");

    private String text;

    /**
     * Constructor.
     * @param text the text for the directive.
     */
    Directive(String text)
    {
        this.text = text;
    }

    /**
     * @return the string that is emitted.
     */
    public String toString()
    {
        return text;
    }
}
```

You'll encounter other directives when you examine complete Jasmin programs.

shows the `CodeGenerator` methods that emit directives.

Listing 15-3: Methods of class `CodeGenerator` that emit Jasmin directives

```
    /**
     * Emit a directive.
     * @param directive the directive code.
     */
    protected void emitDirective(Directive directive)
    {
        assemblyFile.println(directive.toString());
        assemblyFile.flush();
        ++instructionCount;
    }

    /**
     * Emit a 1-operand directive.
     * @param directive the directive code.
     * @param operand the directive operand.
     */
    protected void emitDirective(Directive directive, String operand)
    {
        assemblyFile.println(directive.toString() + " " + operand);
        assemblyFile.flush();
        ++instructionCount;
    }

    /**
     * Emit a 1-operand directive.
     * @param directive the directive code.
     * @param operand the directive operand.
     */
    protected void emitDirective(Directive directive, int operand)
    {
        assemblyFile.println(directive.toString() + " " + operand);
        assemblyFile.flush();
        ++instructionCount;
    }

    /**
     * Emit a 2-operand directive.
     * @param directive the directive code.
     * @param operand the operand.
     */
    protected void emitDirective(Directive directive,
                                 String operand1, String operand2)
    {
        assemblyFile.println(directive.toString() + " " + operand1 +
                                                " " + operand2);
        assemblyFile.flush();
        ++instructionCount;
```

```
    }

    /**
     * Emit a 3-operand directive.
     * @param directive the directive code.
     * @param operand the operand.
     */
    protected void emitDirective(Directive directive,
                                 String operand1, String operand2,
                                 String operand3)
    {
        assemblyFile.println(directive.toString() + " " + operand1 +
                                                    " " + operand2 +
                                                    " " + operand3);
        assemblyFile.flush();
        ++instructionCount;
    }
```

Statement Labels

A Jasmin statement label appears on a separate line and consists of an identifier followed by a colon. An instruction statement preceded by a statement label can be the target of a branch instruction, such as GOTO.

An example of a label is:

```
L001:
```

[Listing 15-4](#) shows class Label in package backend.compiler.

[Listing 15-4:](#) Class `Label`

```
package wci.backend.compiler;

import wci.intermediate.*;

/**
 * <h1>Label</h1>
 *
 * <p>Jasmin instruction label.</p>
 */
public class Label
{
    private static int index = 0;  // index for generating label strings
    private String label;          // the label string

    /**
     * Constructor.
     */
    private Label()
    {
        this.label = "L" + String.format("%03d", ++index);
    }

    /**
     * @return a new instruction label.
     */
    public static Label newLabel()
    {
        return new Label();
    }

    /**
     * @return the label string.
     */
    public String toString()
    {
        return this.label;
    }
}
```

Jasmin labels that the compiler will generate have the form L*xxx*: where *xxx* starts with 001 and increments by 1 for each new label.

[Listing 15-5](#) shows the CodeGenerator methods that emit labels.

[Listing 15-5:](#) Methods of class `CodeGenerator` that emit Jasmin labels

```
    /**
     * Emit a label.
     * @param label the label.
     */
    protected void emitLabel(Label label)
    {
        assemblyFile.println(label + ":");
        assemblyFile.flush();
    }

    /**
     * Emit a label preceded by an integer value for a switch table.
     * @param label the label.
     */
    protected void emitLabel(int value, Label label)
    {
        assemblyFile.println("\t  " + value + ": " + label);
        assemblyFile.flush();
    }

    /**
     * Emit a label preceded by a string value for a switch table.
     * @param label the label.
     */
    protected void emitLabel(String value, Label label)
    {
        assemblyFile.println("\t  " + value + ": " + label);
        assemblyFile.flush();
    }
```

Type Descriptors

Many Jasmin instructions have explicit operands that specify data types. These instructions use Jasmin type descriptors. The type descriptors for the scalar types are:

Java scalar type	Jasmin type descriptor
int	I
float	F
boolean	Z
char	C

The type descriptor `V` represents `void`.

The type descriptor of a class consists of the upper-case letter `L` followed immediately by the fully qualified class name (using `/` instead of `.`) followed immediately by a semicolon `;`.

Some examples:

Java class	Jasmin type descriptor
java.lang.String	Ljava/lang/String;
java.util.HashMap	Ljava/util/HashMap;
Newton	LNewton;

A type descriptor for an array type is prefaced by a left bracket `[` for each array dimension. Some examples:

Java array type	Jasmin type descriptor
java.lang.String[]	[Ljava/lang/String;
Newton[][]	[[LNewton;
int[][][]	[[[I

Another example: A call to the static `format()` method of the Java `String` class has two parameters, the format string and an array of object values, and it returns a string value.

```
String.format("The square root of %4d is %8.4f", n, root)
```

This compiles to the Jasmin instruction

```
invokestatic java/lang/String/format(Ljava/lang/String;[Ljava/lang/Object;)
                                      Ljava/lang/String;
```

This instruction requires that the address of the string constant `"The square root of %4d is %8.4f"` and the address of the array containing the values of `n` and `root` be at the top of the operand stack. The instruction pops off the string and array addresses to be used as parameters of the call, and the call pushes the string value result onto the operand stack.

Listing 15-6 shows methods `typeDescriptor()` and `javaTypeDescriptor()` of class `CodeGenerator`. The latter method is called only by other `CodeGenerator` methods.

Listing 15-6: Methods `typeDescriptor()` and `javaTypeDescriptor()` of class `CodeGenerator`

```
/**
 * Generate a type descriptor of an identifier&apos;s type.
 * @param id the symbol table entry of an identifier.
 * @return the type descriptor.
 */
protected String typeDescriptor(SymTabEntry id)
{
    TypeSpec type = id.getTypeSpec();

    if (type != null) {
        if (isWrapped(id)) {
            return "L" + varParmWrapper(type.baseType()) + ";";
        }
        else {
            return typeDescriptor(id.getTypeSpec());
        }
    }
    else {
        return "V";
    }
}

/**
 * Generate a type descriptor for a data type.
 * @param type the data type.
 * @return the type descriptor.
 */
protected String typeDescriptor(TypeSpec type)
{
    TypeForm form = type.getForm();
    StringBuffer buffer = new StringBuffer();

    while ((form == ARRAY) && !type.isPascalString()) {
        buffer.append("[");
        type = (TypeSpec) type.getAttribute(ARRAY_ELEMENT_TYPE);
        form = type.getForm();
    }

    type = type.baseType();

    if (type == Predefined.integerType) {
        buffer.append("I");
    }
    else if (type == Predefined.realType) {
        buffer.append("F");
    }
    else if (type == Predefined.booleanType) {
```

```
            buffer.append("Z");
        }
        else if (type == Predefined.charType) {
            buffer.append("C");
        }
        else if (type.isPascalString()) {
            buffer.append("Ljava/lang/StringBuilder;");
        }
        else if (form == ENUMERATION) {
            buffer.append("I");
        }
        else /* (form == RECORD) */ {
            buffer.append("Ljava/util/HashMap;");
        }

        return buffer.toString();
    }

    /**
     * Generate a Java type descriptor for a data type.
     * @param type the data type.
     * @return the type descriptor.
     */
    private String javaTypeDescriptor(TypeSpec type)
    {
        TypeForm form = type.getForm();
        StringBuffer buffer = new StringBuffer();
        boolean isArray = false;

        while ((form == ARRAY) && !type.isPascalString()) {
            buffer.append("[");
            type = (TypeSpec) type.getAttribute(ARRAY_ELEMENT_TYPE);
            form = type.getForm();
            isArray = true;
        }

        if (isArray) {
            buffer.append("L");
        }

        type = type.baseType();

        if (type == Predefined.integerType) {
            buffer.append("java/lang/Integer");
        }
        else if (type == Predefined.realType) {
            buffer.append("java/lang/Float");
        }
        else if (type == Predefined.booleanType) {
            buffer.append("java/lang/Boolean");
        }
        else if (type == Predefined.charType) {
            buffer.append("java/lang/Character");
        }
        else if (type.isPascalString()) {
            buffer.append("java/lang/StringBuilder");
        }
        else if (form == ENUMERATION) {
          buffer.append("java/lang/Integer");
        }
        else /* (form == RECORD) */ {
            buffer.append("java/util/HashMap");
        }

        if (isArray) {
            buffer.append(";");
        }

        return buffer.toString();
    }
}
```

The "wrapping" mentioned in the first version of method `typeDescriptor()` is a way for Jasmin to pass scalar variables by reference when executing Pascal `VAR` parameters. This will be explained in greater detail in the next chapter.

Program Structure

The compiler will compile a Pascal program into a Jasmin class. The main body becomes the main method, and each program variable (declared at nesting level 1) becomes a static field of the class. Each procedure or function becomes a static method of the class.

Listing 15-7a shows a simple Pascal program `HelloOnce.pas`. The Pascal compiler will compile that Pascal program and generate code similar to what the Java compiler generates when compiling the Java class `HelloOnce` shown in Listing 15-7b. Listing 15-7c shows the Jasmin assembly object file `helloonce.j` the compiler generates when compiling `HelloOnce.pas`.

Design Note

Since you will compile each Pascal program into a Jasmin program, and Jasmin is the assembly language of the Java Virtual Machine, it makes sense to compare a Pascal program to an equivalent Java program.

Listing 15-7a: A simple Pascal program `HelloOnce.pas`

```
PROGRAM HelloOnce;

BEGIN
    writeln(&apos;Hello, world.&apos;)
END.
```

Listing 15-7b: The Pascal compiler compiles `HelloOnce.pas` and generates object code similar to what the Java compiler would generate when it compiles this `HelloOnce` Java class.

```
public class HelloOnce
{
    private static RunTimer     _runTimer;
    private static PascalTextIn _standardIn;

    public static void main(String[] args)
    {
        _runTimer = new RunTimer();
        _standardIn = new PascalTextIn();

        System.out.println("Hello, world.");

        _runTimer.printElapsedTime();
    }
}
```

Listing 15-7c: The Jasmin assembly object file `helloonce.j` generated by the Pascal compiler when it compiles `HelloOnce.pas`

```
.class public helloonce
.super java/lang/Object

.field private static _runTimer LRunTimer;
.field private static _standardIn LPascalTextIn;

.method public <init>()V

    aload_0
    invokenonvirtual  java/lang/Object/<init>()V
    return

.limit locals 1
.limit stack 1
.end method

.method public static main([Ljava/lang/String;)V

    new      RunTimer
    dup
    invokenonvirtual  RunTimer/<init>()V
    putstatic         helloonce/_runTimer LRunTimer;
    new      PascalTextIn
    dup
    invokenonvirtual  PascalTextIn/<init>()V
    putstatic         helloonce/_standardIn LPascalTextIn;

.line 4
    getstatic         java/lang/System/out Ljava/io/PrintStream;
    ldc      "Hello, world.\n"
    invokevirtual     java/io/PrintStream.print(Ljava/lang/String;)V

    getstatic         helloonce/_runTimer LRunTimer;
    invokevirtual     RunTimer.printElapsedTime()V

    return

.limit locals 1
.limit stack 3
.end method
```

As described in the next chapter, the Pascal compiler adds two static fields, `_runTimer` and `_standardIn`. Their names begin with an underscore to prevent name clashes with any Pascal variables. Field `_runTimer` keeps track of the program's execution time, and field `_standardIn` provides runtime input. The Java class `HelloOnce` in Listing 15-7b includes these two fields.

Classes `RunTimer` and `PascalTextIn` are contained in the Pascal Runtime Library, an archive file `PascalRTL.jar` that you'll develop in the following chapters.

Listings 15-8a shows a slightly more complex Pascal program: `HelloMany.pas`.

Listing 15-8a: Pascal program `HelloMany.pas`

```
PROGRAM HelloMany;

VAR
    count : integer;

PROCEDURE sayHello(howManyTimes : integer);

    VAR
        i : integer;

    BEGIN
        FOR i := 1 TO howManyTimes DO BEGIN
            writeln(i:3, &apos;: &apos;, &apos;Hello, world.&apos;)
        END
    END;

BEGIN {HelloMany}
    write(&apos;How many times? &apos;);
    writeln;

    sayHello(count);
END.
```

Listing 15-8b shows the "equivalent" Java class `HelloMany`.

Listing 15-8b: The "equivalent" Java class `HelloMany`

```
public class HelloMany
{
    private static RunTimer     _runTimer;
    private static PascalTextIn _standardIn;
```

```
    private static int count;

    private static void sayHello(int howManyTimes)
    {
        int i;

        for (i = 1; i <= howManyTimes; ++i) {
            System.out.print(String.format("%3d: Hello, world.\n", i));
        }
    }

    public static void main(String[] args)
        throws Exception
    {
        _runTimer = new RunTimer();
        _standardIn = new PascalTextIn();

        System.out.print("How many times? ");
        count = _standardIn.readInteger();
        System.out.println();

        sayHello(count);

        _runTimer.printElapsedTime();
    }
}
```

   Listing 15-8c shows the Jasmin assembly object file `hellomany.j` generated by the Pascal compiler when it compiles
`HelloMany.pas`.

Listing 15-8c: **The Jasmin assembly object file `hellomany.j` generated by the Pascal compiler when it compiles `HelloMany.pas`**

```
.class public hellomany
.super java/lang/Object

.field private static _runTimer LRunTimer;
.field private static _standardIn LPascalTextIn;

.field private static count I

.method public <init>()V

        aload_0
        invokenonvirtual  java/lang/Object/<init>()V
        return

.limit locals 1
.limit stack 1
.end method

.method private static sayhello(I)V

.var 0 is howmanytimes I
.var 1 is i I

.line 12
        iconst_1
        istore_1
.line 12
L001:
        iload_1
        iload_0
        if_icmpgt       L003
        iconst_0
        goto        L004
L003:
        iconst_1
L004:
        ifne        L002
.line 13
        getstatic         java/lang/System/out Ljava/io/PrintStream;
        ldc     "%3d: Hello, world.\n"
        iconst_1
        anewarray         java/lang/Object
        dup
        iconst_0
        iload_1
        invokestatic    java/lang/Integer.valueOf(I)Ljava/lang/Integer;
aastore
invokestaticjava/lang/String/format(Ljava/lang/String;[Ljava/lang/
                                      Object;)Ljava/lang/String;
invokevirtualjava/io/PrintStream.print(Ljava/lang/String;)V
.line 12
        iload_1
        iconst_1
        iadd
        istore_1
        goto        L001
L002:

        return

.limit locals 2
.limit stack 7
.end method

.method public static main([Ljava/lang/String;)V

        new       RunTimer
        dup
        invokenonvirtual  RunTimer/<init>()V
        putstatic         hellomany/_runTimer LRunTimer;
        new       PascalTextIn
        dup
        invokenonvirtual  PascalTextIn/<init>()V
        putstatic         hellomany/_standardIn LPascalTextIn;

.line 18
        getstatic         java/lang/System/out Ljava/io/PrintStream;
        ldc     "How many times? "
        invokevirtual     java/io/PrintStream.print(Ljava/lang/String;)V
.line 19
        getstatic         hellomany/_standardIn LPascalTextIn;
        invokevirtual     PascalTextIn.readInteger()I
        putstatic         hellomany/count I
.line 20
```

```
        getstatic        java/lang/System/out Ljava/io/PrintStream;
        invokevirtual    java/io/PrintStream.println()V
.line 22
        getstatic        hellomany/count I
        invokestatic     hellomany/sayhello(I)V

        getstatic        hellomany/_runTimer LRunTimer;
        invokevirtual    RunTimer.printElapsedTime()V

        return

.limit locals 1
.limit stack 3
.end method
```

Each Jasmin statement must fit in one text line. One line, the second `invokestatic` instruction statement after the `.line 13` directive, is too long to show in as a single line. Therefore, the listing shows the latter part of the statement on the next line right-justified to indicate that it's a continuation of the preceding line with no intervening blank. In the `hellomany.j` file, it's a single line.

Programs and Fields

As shown in Listings and , the compiler will compile each Pascal program as if it were a Java class. The generated object code includes the `<init>` method which is the default (parameterless) class constructor. The main body of the Pascal program is compiled as if it were the `main()` method of the class, and each program variable (declared at nesting level 1) is compiled into a private static field of the class.

Pascal program `HelloMany.pas` has one program variable `count`. Both `HelloOnce.pas` and `HelloMany.pas` also have the generated program variables `_runTimer` and `_standardIn`. Each field in the Jasmin object program is specified by a `.field` directive.

Methods

The Pascal compiler will compile each Pascal procedure or function as the Java compiler would compile a private static method of a class. Therefore, procedure `sayHello` in will be compiled as the Java compiler would compile private static method `sayHello()` in .

Jasmin methods have local variables, such as formal parameter `howManyTimes` and declared variable `i`. As described above, local variable values are kept in slots of the local variables array in the method's stack frame. The `.var` directive indicates the slot number, name, and data type of each local variable. This directive is optional but is useful for debugging.

Sizes of the Local Variables Array and the Operand Stack

The Pascal compiler must keep track of the maximum sizes of the local variables array and of the operand stack as it compiles the main program and each procedure or function. It does that with the aid of two classes: `LocalVariables` and `LocalStack` in package `backend.compiler`.

shows class `LocalVariables`.

Class `LocalVariables`

```java
package wci.backend.compiler;

import java.util.ArrayList;

/**
 * <h1>LocalVariables</h1>
 *
 * <p>Maintain a method&apos;s local variables array.</p>
 */
public class LocalVariables
{
    // List of booleans to keep track of reserved local variables. The ith
    // element is true if the ith variable is being used, else it is false.
    // The final size of the list is the total number of local variables
    // used by the method.
    private ArrayList<Boolean> reserved;

    /**
     * Constructor.
     * @param index initially reserve local variables 0 through index-1.
     */
    public LocalVariables(int index)
    {
        reserved = new ArrayList<Boolean>();

        for (int i = 0; i <= index; ++i) {
            reserved.add(true);
        }
    }

    /**
     * Reserve a local variable.
     * @return the index of the newly reserved variable.
     */
    public int reserve()
    {
        // Search for existing but unreserved local variables.
        for (int i = 0; i < reserved.size(); ++i) {
            if (! reserved.get(i)) {
                reserved.set(i, true);
                return i;
            }
        }

        // Reserved a new variable.
        reserved.add(true);
        return reserved.size()-1;
    }

    /**
     * Release a local variable that&apos;s no longer needed.
     * @param index the index of the variable.
```

```
     */
    public void release(int index)
    {
        reserved.set(index, false);
    }

    /**
     * Return the count of local variables needed by the method.
     * @return the count.
     */
    public int count()
    {
        return reserved.size();
    }
}
```

The `LocalVariables` constructor will be called to make an initial allocation for each Jasmin method corresponding to the Pascal main program or a procedure or function. The array list `reserved` keeps track of which slots of the local variables array are allocated.

Method `reserve()` allocates a free slot and it will often be called to allocate a temporary variable. It searches the existing slots and adds a new one if necessary. At the end of compiling a Pascal main program, procedure, or function, the size of array list `reserved` is the count of slots that the Jasmin method will need at run time, and method `count()` returns this value. Method `release()` frees a slot.

Listing 15-10 shows class `LocalStack`.

Listing 15-10: Class `LocalStack`

```
package wci.backend.compiler;

/**
 * <h1>LocalStack</h1>
 *
 * <p>Maintain a method&apos;s local runtime stack.</p>
 */
public class LocalStack
{
    private int size;     // current stack size
    private int maxSize;  // maximum attained stack size

    /**
     * Constructor
     * @param size initial stack size.
     */
    public LocalStack()
    {
        this.size = 0;
        this.maxSize = 0;
    }

    /**
     * Getter
     * @return the current stack size.
     */
    public int getSize()
    {
        return this.size;
    }

    /**
     * Increase the stack size by a given amount.
     * @param amount the amount to increase.
     */
    public void increase(int amount)
    {
        size += amount;
        maxSize = Math.max(maxSize, size);
    }

    /**
     * Decrease the stack size by a given amount.
     * @param amount the amount to decrease.
     */
    public void decrease(int amount)
    {
        size -= amount;
    }

    /**
     * Increase and decrease the stack size by the same amount.
     * @param amount the amount to increase and decrease.
     */
    public void use(int amount)
    {
        increase(amount);
        decrease(amount);
    }

    /**
     * Increase and decrease the stack size by the different amounts.
     * @param amountIncrease the amount to increase.
     * @param amountDecrease the amount to decrease.
     */
    public void use(int amountIncrease, int amountDecrease)
    {
        increase(amountIncrease);
        decrease(amountDecrease);
    }

    /**
     * Return the maximum attained stack size.
     * @return the maximum size.
     */
    public int capacity()
    {
        return maxSize;
    }
}
```

The `LocalStack` constructor will be called at the start of compiling the Pascal main program and each procedure and function. As the compiler generates Jasmin instructions, the `LocalStack` methods keep track of the current size of the operand stack when those instructions are executed at run time. Field `size` is the current stack size, and field `maxSize` is the tallest the stack ever gets when the main program or each procedure or function is executed.

Method `increase()` increases the current stack size and adjusts `maxSize` if necessary. This may be called by the compiler, for example, after emitting load instructions to push values onto the operand stack. Method `decrease()` deceases the current stack size. The two `use()` methods are convenience methods to increase the stack size and then decrease it with a single call. Method `capacity()` will be called at the end of compiling a Pascal main program, procedure, or function to return the tallest the operand stack will get while executing the Jasmin method.

At the end of compiling a Pascal main program, procedure, or function as a Jasmin method, the Pascal compiler must generate `.limit locals` and `.limit stack` directives in the Jasmin assembly object code. For example:

```
.limit locals 2
.limit stack 7
```

See also Listings 15-6c and 15-7c. These directives tell how many slots for the local variables array and how many operand stack elements to allocate, respectively, for the Jasmin method at run time. As you just saw, the code generator can obtain these values by calling `localVariables.count()` and `localStack.capacity()`.

Emitting Instructions

Listing 15-11 shows the enumerated type `Instruction` in package `backend.compiler`. It represents all the Jasmin operation mnemonics that the Pascal compiler can emit as it generates instruction statements.[2]

[2] Jasmin has other instructions that the compiler won't emit in this book, including instructions that work with long and double data types.

Listing 15-11: Enumerated type `Instruction`

```
package wci.backend.compiler;

/**
 * <h1>Instruction</h1>
 *
 * <p>Jasmin instructions.</p>
 */
public enum Instruction
{
    // Load constant
    ICONST_0, ICONST_1, ICONST_2, ICONST_3, ICONST_4, ICONST_5, ICONST_M1,
    FCONST_0, FCONST_1, FCONST_2, ACONST_NULL,
    BIPUSH, SIPUSH, LDC,

    // Load value or address
    ILOAD_0, ILOAD_1, ILOAD_2, ILOAD_3,
    FLOAD_0, FLOAD_1, FLOAD_2, FLOAD_3,
    ALOAD_0, ALOAD_1, ALOAD_2, ALOAD_3,
    ILOAD, FLOAD, ALOAD,
    GETSTATIC, GETFIELD,

    // Store value or address
    ISTORE_0, ISTORE_1, ISTORE_2, ISTORE_3,
    FSTORE_0, FSTORE_1, FSTORE_2, FSTORE_3,
    ASTORE_0, ASTORE_1, ASTORE_2, ASTORE_3,
    ISTORE, FSTORE, ASTORE,
    PUTSTATIC, PUTFIELD,

    // Arithmetic and logical
    IADD, LADD, FADD, ISUB, LSUB, FSUB, IMUL, LMUL, FMUL,
    IDIV, LDIV, FDIV, IREM, LREM, FREM, INEG, FNEG,
    IINC, IAND, IOR, IXOR,

    // Type conversion and checking
    I2F, I2C, I2D, F2I, F2D, D2F,
    CHECKCAST,

    // Compare and branch
    IFEQ, IFNE, IFLT, IFLE, IFGT, IFGE,
    IF_ICMPEQ, IF_ICMPNE, IF_ICMPLT, IF_ICMPLE, IF_ICMPGT, IF_ICMPGE,
    FCMPG, GOTO, LOOKUPSWITCH, LOOKUPTABLE,

    // Objects and arrays
    NEW, NEWARRAY, ANEWARRAY, MULTIANEWARRAY,
    IALOAD, FALOAD, BALOAD, CALOAD, AALOAD,
    IASTORE, FASTORE, BASTORE, CASTORE, AASTORE,

    // Call and return
    INVOKESTATIC, INVOKEVIRTUAL, INVOKENONVIRTUAL,
    RETURN, IRETURN, FRETURN, ARETURN,

    // Operand stack
    POP, SWAP, DUP,

    // No operation
    NOP;

    /**
     * @return the string that is emitted.
     */
    public String toString()
    {
        return super.toString().toLowerCase();
    }
}
```

Listing 15-12 shows the methods in class `CodeGenerator` that emit instructions with zero operands or with one or two operands of various types.

Listing 15-12: Methods that emit instructions in class `CodeGenerator`

```
    /**
```

```
 * Emit a 0-operand instruction.
 * @param opcode the operation code.
 */
protected void emit(Instruction opcode)
{
    assemblyFile.println("\t" + opcode.toString());
    assemblyFile.flush();
    ++instructionCount;
}

/**
 * Emit a 1-operand instruction.
 * @param opcode the operation code.
 * @param operand the operand text.
 */
protected void emit(Instruction opcode, String operand)
{
    assemblyFile.println("\t" + opcode.toString() + "\t" + operand);
    assemblyFile.flush();
    ++instructionCount;
}

/**
 * Emit a 1-operand instruction.
 * @param opcode the operation code.
 * @param operand the operand value.
 */
protected void emit(Instruction opcode, int operand)
{
    assemblyFile.println("\t" + opcode.toString() + "\t" + operand);
    assemblyFile.flush();
    ++instructionCount;
}

/**
 * Emit a 1-operand instruction.
 * @param opcode the operation code.
 * @param operand the operand value.
 */
protected void emit(Instruction opcode, float operand)
{
    assemblyFile.println("\t" + opcode.toString() + "\t" + operand);
    assemblyFile.flush();
    ++instructionCount;
}

/**
 * Emit a 1-operand instruction.
 * @param opcode the operation code.
 * @param label the label operand.
 */
protected void emit(Instruction opcode, Label label)
{
    assemblyFile.println("\t" + opcode.toString() + "\t" + label);
    assemblyFile.flush();
    ++instructionCount;
}

/**
 * Emit a 2-operand instruction.
 * @param opcode the operation code.
 * @param operand1 the value of the first operand.
 * @param operand2 the value of the second operand.
 */
protected void emit(Instruction opcode, int operand1, int operand2)
{
    assemblyFile.println("\t" + opcode.toString() +
                            "\t" + operand1 + " " + operand2);
    assemblyFile.flush();
    ++instructionCount;
}

/**
 * Emit a 2-operand instruction.
 * @param opcode the operation code.
 * @param operand1 the text of the first operand.
 * @param operand2 the text of the second operand.
 */
protected void emit(Instruction opcode, String operand1, String operand2)
{
    assemblyFile.println("\t" + opcode.toString() +
                            "\t" + operand1 + " " + operand2);
    assemblyFile.flush();
    ++instructionCount;
}
```

Load and Store Instructions

Jasmin has instructions to load (push a value onto the operand stack) an integer, floating-point, or string constant. It has instructions to load a value from a local variable, field, or array element. It has instructions to store (pop off a value from the operand stack to save it) a value into a local variable, field, or array element.

Load Constant

The basic instruction to load a constant is LDC, which has one explicit operand. Examples:

```
ldc 12
ldc 1.0E-6
ldc "The square root of %4d is %8.4f\n"
```

Note that when a string constant is loaded, as in the preceding example, the address of the string constant is pushed onto the operand stack.

Jasmin has several "shortcut" instructions that load small integer values: ICONST_0, ICONST_1, ICONST_2, ICONST_3, ICONST_4, ICONST_5, and, ICONST_M1 load the values 0, 1, 2, 3, 4, 5, and -1, respectively. Each instruction has no explicit operand and is shorter (and likely faster) than the equivalent LDC instruction. Similarly, the instructions FCONST_0, FCONST_1, FCONST_2 load the small floating-point values 0, 1, and 2, respectively.

The instruction BIPUSH loads an integer value from -128 up to and including 127, and the instruction SIPUSH loads an integer value from -32,768 up to and including 32,767. Each has a single explicit operand. Examples:

```
      bipush 14
      sipush 1024
```

The instruction `ACONST_NULL` has no explicit operand and loads the null value.

Several `emitLoadConstant()` methods of class `CodeGenerator` emit instructions to load a constant. See Listing 15-13.

Listing 15-13: Methods that emit instructions to load a constant in class `CodeGenerator`

```java
/**
 * Emit a load of an integer constant value.
 * @param value the constant value.
 */
protected void emitLoadConstant(int value)
{
    switch (value) {
        case -1: emit(ICONST_M1); break;
        case  0: emit(ICONST_0);  break;
        case  1: emit(ICONST_1);  break;
        case  2: emit(ICONST_2);  break;
        case  3: emit(ICONST_3);  break;
        case  4: emit(ICONST_4);  break;
        case  5: emit(ICONST_5);  break;

        default: {
            if ((-128 <= value) && (value <= 127)) {
                emit(BIPUSH, value);
            }
            else if ((-32768 <= value) && (value <= 32767)) {
                emit(SIPUSH, value);
            }
            else {
                emit(LDC, value);
            }
        }
    }
}

/**
 * Emit a load of a real constant value.
 * @param value the constant value.
 */
protected void emitLoadConstant(float value)
{
    if (value == 0.0f) {
        emit(FCONST_0);
    }
    else if (value == 1.0f) {
        emit(FCONST_1);
    }
    else if (value == 2.0f) {
        emit(FCONST_2);
    }
    else {
        emit(LDC, value);
    }
}

/**
 * Emit a load of a string constant value.
 * @param value the constant value.
 */
protected void emitLoadConstant(String value)
{
    emit(LDC, "\"" + value + "\"");
}
```

Load Local Value

A local value of a Jasmin method is kept in the local variables array of the method's stack frame. The basic instructions to load values are `ILOAD`, `FLOAD`, and `ALOAD` to load an integer, floating-point, and address value, respectively. Each has one explicit operand, the index number of the value's slot in the local variables array of the active stack frame. The `ILOAD` instruction also loads a boolean or character value. Examples:

```
      iload 7
      fload 12
      aload 4
```

Note that in each instruction above, the operand is the *slot number* of the value to load from the local variables array.

There are shortcut instructions to load integer, floating-point, and address values from slots 0, 1, 2, and 3: `ILOAD_0`, `ILOAD_1`, `ILOAD_2`, `ILOAD_3`, `FLOAD_0`, `FLOAD_1`, `FLOAD_2`, `FLOAD_3`, `ALOAD_0`, `ALOAD_1`, `ALOAD_2`, and `ALOAD_3`. Since the slot number is "built into" the instruction mnemonic, each instruction has no explicit operands.

Listing 15-14 shows method `emitLoadLocal()`.

Listing 15-14: Method `emitLoadLocal()` of class `CodeGenerator`

```java
/**
 * Emit a load instruction for a local variable.
 * @param type the variable's data type.
 * @param index the variable's index into the local variables array.
 */
protected void emitLoadLocal(TypeSpec type, int index)
{
    TypeForm form = null;

    if (type != null) {
        type = type.baseType();
        form = type.getForm();
    }

    if ((type == Predefined.integerType) ||
        (type == Predefined.booleanType) ||
```

```
                    (type == Predefined.charType)    ||
                    (form == ENUMERATION))
            {
                switch (index) {
                    case 0:  emit(ILOAD_0); break;
                    case 1:  emit(ILOAD_1); break;
                    case 2:  emit(ILOAD_2); break;
                    case 3:  emit(ILOAD_3); break;
                    default: emit(ILOAD, index);
                }
            }
            else if (type == Predefined.realType) {
                switch (index) {
                    case 0:  emit(FLOAD_0); break;
                    case 1:  emit(FLOAD_1); break;
                    case 2:  emit(FLOAD_2); break;
                    case 3:  emit(FLOAD_3); break;
                    default: emit(FLOAD, index);
                }
            }
            else {
                switch (index) {
                    case 0:  emit(ALOAD_0); break;
                    case 1:  emit(ALOAD_1); break;
                    case 2:  emit(ALOAD_2); break;
                    case 3:  emit(ALOAD_3); break;
                    default: emit(ALOAD, index);
                }
            }
        }
    }
```

Load Variable

Jasmin instruction GETSTATIC loads the value of a static field of a class. (Recall that the Pascal compiler will generate a static field for each level-1 program variable.) It requires two explicit operands: the fully qualified name of the field (with a slash / as name separator) and the type descriptor of the field. Examples from :

```
getstatic          hellomany/count I
getstatic          hellomany/_standardIn LPascalTextIn;
getstatic          java/lang/System/out Ljava/io/PrintStream;
```

Jasmin instruction GETFIELD loads a value from an object field. The Pascal compiler will use this instruction to load a wrapped scalar value that was passed as a VAR parameter, as explained in the next chapter. It has two explicit operands similar to those of instruction GETSTATIC. Example:

```
getfield  IWrap/value I
```

shows method emitLoadVariable().

Listing 15-15: Method emitLoadVariable() of class CodeGenerator

```
/**
 * Emit code to load the value of a variable, which can be
 * a program variable, a local variable, or a VAR parameter.
 * @param variableId the variable&apos;s symbol table entry.
 */
protected void emitLoadVariable(SymTabEntry variableId)
{
    TypeSpec variableType = variableId.getTypeSpec().baseType();
    int nestingLevel = variableId.getSymTab().getNestingLevel();

    // Program variable.
    if (nestingLevel == 1) {
        String programName = symTabStack.getProgramId().getName();
        String variableName = variableId.getName();
        String name = programName + "/" + variableName;

        emit(GETSTATIC, name, typeDescriptor(variableType));
    }

    // Wrapped variable.
    else if (isWrapped(variableId)) {
        int index = (Integer) variableId.getAttribute(INDEX);
        emitLoadLocal(null, index);
        emit(GETFIELD, varParmWrapper(variableType) + "/value",
                       typeDescriptor(variableType));
    }

    // Local variable.
    else {
        int index = (Integer) variableId.getAttribute(INDEX);
        emitLoadLocal(variableType, index);
    }
}
```

Load Array Element

Jasmin instructions IALOAD, FALOAD, BALOAD, CALOAD, and AALOAD load an integer, floating-point, boolean, character, and address value, respectively, from an array element. Each has no explicit operands, but requires two operand values on top of the operand stack, the address of the array and the integer value of the element index.

If the array is actually a string, the Pascal compiler emits the INVOKEVIRTUAL instruction to call the charAt() method of java.lang.StringBuilder, which returns a character value. For example:

```
invokevirtual    java/lang/StringBuilder.charAt(I)C
```

Method emitLoadArrayElement() emits instructions to load array element values, as shown in .

Listing 15-16: Method emitLoadArrayElement() of class CodeGenerator

```
/**
```

```
 * Emit a load of an array element.
 * @param elmtType the element type if character, else null.
 */
protected void emitLoadArrayElement(TypeSpec elmtType)
{
    TypeForm form = null;

    if (elmtType != null) {
        elmtType = elmtType.baseType();
        form = elmtType.getForm();
    }

    // Load a character from a string.
    if (elmtType == Predefined.charType) {
        emit(INVOKEVIRTUAL, "java/lang/StringBuilder.charAt(I)C");
    }

    // Load an array element.
    else {
        emit(  elmtType == Predefined.integerType ? IALOAD
             : elmtType == Predefined.realType    ? FALOAD
             : elmtType == Predefined.booleanType ? BALOAD
             : elmtType == Predefined.charType    ? CALOAD
             : form == ENUMERATION                ? IALOAD
             :                                      AALOAD);
    }
}
```

Store Local Value

Jasmin's `ISTORE`, `FSTORE`, and `ASTORE` instructions pop an integer, floating-point, and address value, respectively, off the operand stack and store it into a slot of the local variables array of the active stack frame. Each instruction has one explicit operand, the slot number, and of course each one also expects a value of the appropriate type on top of the operand stack. The `ISTORE` instruction also stores a boolean or character value. Examples:

```
istore  7
fstore  12
astore  6
```

As with the load instructions, there are the shortcut instructions to store integer, floating-point, and address values into slots 0, 1, 2, and 3: `ISTORE_0`, `ISTORE_1`, `ISTORE_2`, `ISTORE_3`, `FSTORE_0`, `FSTORE_1`, `FSTORE_2`, `FSTORE_3`, `ASTORE_0`, `ASTORE_1`, `ASTORE_2`, and `ASTORE_3`. Each one has no explicit operands but expects a value of the appropriate type on top of the operand stack.

Listing 15-17 shows method `emitStoreLocal()`.

Listing 15-17: Method `emitStoreLocal()` of class `CodeGenerator`

```
/**
 * Emit a store instruction into a local variable.
 * @param type the data type of the variable.
 * @param index the local variable index.
 */
protected void emitStoreLocal(TypeSpec type, int index)
{
    TypeForm form = null;

    if (type != null) {
        type = type.baseType();
        form = type.getForm();
    }

    if ((type == Predefined.integerType) ||
        (type == Predefined.booleanType) ||
        (type == Predefined.charType)    ||
        (form == ENUMERATION))
    {
        switch (index) {
            case 0:  emit(ISTORE_0); break;
            case 1:  emit(ISTORE_1); break;
            case 2:  emit(ISTORE_2); break;
            case 3:  emit(ISTORE_3); break;
            default: emit(ISTORE, index);
        }
    }
    else if (type == Predefined.realType) {
        switch (index) {
            case 0:  emit(FSTORE_0); break;
            case 1:  emit(FSTORE_1); break;
            case 2:  emit(FSTORE_2); break;
            case 3:  emit(FSTORE_3); break;
            default: emit(FSTORE, index);
        }
    }
    else {
        switch (index) {
            case 0:  emit(ASTORE_0); break;
            case 1:  emit(ASTORE_1); break;
            case 2:  emit(ASTORE_2); break;
            case 3:  emit(ASTORE_3); break;
            default: emit(ASTORE, index);
        }
    }
}
```

Store Field

Jasmin instructions `PUTSTATIC` and `PUTFIELD` store a value from the operand stack into a static field and an object field, respectively. Each has two explicit operands: the fully qualified field name and the type descriptor. Examples:

```
putstatic       xref/wordtablefull Z
putstatic       xref/wordtable [Ljava/util/HashMap;
putstatic       strings1/_runTimer LRunTimer;
putstatic       wolfisland/foodunits [[I
```

```
        putfield        IWrap/value I
```

shows the `emitStoreVariable()` methods.

Listing 15-18: The `emitStoreVariable()` methods of class `CodeGenerator`

```
/**
 * Emit code to store a value into a variable, which can be
 * a program variable, a local variable, or a VAR parameter.
 * @param variableId the symbol table entry of the variable.
 */
protected void emitStoreVariable(SymTabEntry variableId)
{
    int nestingLevel = variableId.getSymTab().getNestingLevel();
    int index = (Integer) variableId.getAttribute(INDEX);

    emitStoreVariable(variableId, nestingLevel, index);
}

/**
 * Emit code to store a value into a variable, which can be
 * a program variable, a local variable, or a VAR parameter.
 * @param variableId the symbol table entry of the variable.
 * @param nestingLevel the variable&apos;s nesting level.
 * @param index the vaiable&apos;s index.
 */
protected void emitStoreVariable(SymTabEntry variableId, int nestingLevel,
                                 int index)
{
    TypeSpec variableType = variableId.getTypeSpec();

    // Program variable.
    if (nestingLevel == 1) {
        String targetName = variableId.getName();
        String programName = symTabStack.getProgramId().getName();
        String name = programName + "/" + targetName;

        emitRangeCheck(variableType);
        emit(PUTSTATIC, name, typeDescriptor(variableType.baseType()));
    }

    // Wrapped parameter: Set the wrapper&apos;s value field.
    else if (isWrapped(variableId)) {
        emitRangeCheck(variableType);
        emit(PUTFIELD, varParmWrapper(variableType.baseType()) + "/value",
                       typeDescriptor(variableType.baseType()));
    }

    // Local variable.
    else {
        emitRangeCheck(variableType);
        emitStoreLocal(variableType.baseType(), index);
    }
}
```

The call to `emitRangeCheck()` generates code to do runtime range checking. You will examine range checking in the next chapter.

Store Array Element

Jasmin instructions `IASTORE`, `FASTORE`, `BASTORE`, `CASTORE`, and `AASTORE` store an integer, floating-point, boolean, character, and address value, respectively, into an array element. Each has no explicit operands, but requires three operand values on top of the operand stack: the address of the array, the integer value of the element index, and a value of the appropriate type to store.

If the array is actually a string, the Pascal compiler emits the `INVOKEVIRTUAL` instruction to call the `setCharAt()` method of `java.lang.StringBuilder`, which has no return value:

```
invokevirtual   java/lang/StringBuilder.setCharAt(IC)V
```

shows method `emitStoreArrayElement()`.

Listing 15-19: Method `emitStoreArrayElement()` of class `CodeGenerator`

```
/**
 * Emit a store of an array element.
 * @param elmtType the element type.
 */
protected void emitStoreArrayElement(TypeSpec elmtType)
{
    TypeForm form = null;

    if (elmtType != null) {
        elmtType = elmtType.baseType();
        form = elmtType.getForm();
    }

    if (elmtType == Predefined.charType) {
        emit(INVOKEVIRTUAL, "java/lang/StringBuilder.setCharAt(IC)V");
    }
    else {
        emit(  elmtType == Predefined.integerType ? IASTORE
             : elmtType == Predefined.realType    ? FASTORE
             : elmtType == Predefined.booleanType ? BASTORE
             : elmtType == Predefined.charType    ? CASTORE
             : form == ENUMERATION                ? IASTORE
             :                                      AASTORE);
    }
}
```

Data Manipulation Instructions

Jasmin has numerous instructions that manipulate data values that are on top of the operand stack.

Pop, Swap, and Duplicate

The POP instruction pops off and discards the value at the top of the operand stack. The SWAP instruction exchanges the top two values of the operand stack. The DUP instruction makes a copy of the value at the top of the operand stack and then pushes the copy onto the stack.

Arithmetic

Jasmin instruction IADD has no explicit operands but requires two integer values at the top of the operand stack. It pops off the two values, adds them, and pushes their sum onto the stack. IMUL is similar except that it multiplies the values and pushes their product. ISUB subtracts the topmost value on the operand stack from the value beneath it and pushes their difference. IDIV performs integer division by dividing the topmost value on the operand stack into the value beneath it and pushes the truncated integer quotient. IREM performs integer division but pushes the remainder instead.

Corresponding instructions for floating-point values are FADD, FSUB, FMUL, FDIV, and FREM. Instruction FDIV performs a floating-point division to generate a floating-point quotient.

Instructions INEG and FNEG negate the integer or floating-point value, respectively, at the top of the operand stack. They have no explicit operands.

Instruction IINC increments the integer value of a local variable. It has two explicit operands, the first is the slot number of the local variable, and the second is the integer constant value from -128 up to and including 127, which is the amount to increment the local value.

Logical

Jasmin instructions IAND, IOR, and IXOR pop off the top two integer values from the operand stack, perform a logical and, or, and exclusive or operation on the values, respectively, and pushes the result onto the operand stack. They have no explicit operands.

Convert and Check Type

The Jasmin instructions I2F, I2C, I2D, F2I, F2D, and D2F each converts the type of the value at the top of the operand stack: integer to floating-point, integer to character, floating-point to integer, floating-point to double, and double to floating-point, respectively.[3]

[3] In this book, the Pascal compiler generally will not emit instructions for type double, but some of the functions in `java.lang.Math` require a double parameter value or return a double value.

The CHECKCAST instruction checks that the value on top of the operand stack can be cast to a given type. It has one explicit operand, either the fully qualified name of a class, or the type descriptor of an array. Examples:

```
checkcast  java/lang/Integer
checkcast  java/util/HashMap
checkcast  [[[I
checkcast  [Ljava/lang/StringBuilder;
```

Listing 15-30 shows the emitCheckCastClass() and emitCheckCastClass() methods.

Listing 15-30: Methods `emitCheckCastClass()` and `emitCheckCastClass()` of class `CodeGenerator`

```
/**
 * Emit the CHECKCAST instruction for a scalar type.
 * @param type the data type.
 */
protected void emitCheckCast(TypeSpec type)
{
    String descriptor = typeDescriptor(type);

    // Don't bracket the type with L; if it's not an array.
    if (descriptor.charAt(0) == 'L') {
        descriptor = descriptor.substring(1, descriptor.length()-1);
    }

    emit(CHECKCAST, descriptor);
}

/**
 * Emit the CHECKCAST instruction for a class.
 * @param type the data type.
 */
protected void emitCheckCastClass(TypeSpec type)
{
    String descriptor = javaTypeDescriptor(type);

    // Don't bracket the type with L; if it's not an array.
    if (descriptor.charAt(0) == 'L') {
        descriptor = descriptor.substring(1, descriptor.length()-1);
    }

    emit(CHECKCAST, descriptor);
}
```

Create Objects and Arrays

Jasmin's NEW instruction creates an object (in the heap) and pushes the object's address onto the operand stack. It has one explicit operand: the fully qualified name of the class. Examples:

```
new  RunTimer
new  PascalTextIn
new  java/util/HashMap
```

Instruction NEWARRAY creates an array of scalar values. It has one explicit operand, the scalar type int, float, boolean, or char, and it expects the integer value on top of the operand stack which is the count of elements to create for the array. It creates the array and pushes the array's address onto the operand stack. Examples:

```
newarray  int
newarray  float
```

Instruction `ANEWARRAY` is similar, except that it creates an array of object references (addresses). It requires the count of elements on top of the operand stack, and its one explicit operand is the fully qualified name of a class. Examples:

```
anewarray  java/lang/Object
anewarray  java/lang/StringBuilder
```

Instruction `MULTIANEWARRAY` creates multidimensional arrays. It has two explicit operands, the array type descriptor and the number of dimensions. For each dimension, the instruction expects the number of elements at the top of the operand stack. So the equivalent of the Java statement

```
new int[2][3][4]
```

is the Jasmin instruction sequence

```
bipush  2
bipush  3
bipush  4
multianewarray  [[[I 3
```

Other examples of `MULTIANEWARRAY`:

```
multianewarray  [[F 2
multianewarray  [[[Ljava/lang/StringBuilder; 3
```

Control Instructions

The Jasmin control instructions perform comparisons and branches, as well as procedure and function calls and returns.

Unconditional Branch

The Jasmin `GOTO` instruction does an unconditional branch to a labeled statement. Its one explicit operand is the statement label (without the colon). For example:

```
goto  L088
```

A `GOTO` instruction can branch only to another statement in the same Jasmin method.

Compare and Branch

The Jasmin instructions `IFEQ`, `IFNE`, `IFLT`, `IFLE`, `IFGT`, and `IFGE` each pops off the integer value at the top of the operand stack, compares the value to zero, and branches if the value is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to zero, respectively. Each instruction has a statement label as its one explicit operand. Examples:

```
ifeq  L077
iflt  L02
ifge  L020
```

A common Jasmin programming convention is integer value 0 represents the boolean value false and 1 represents true.

The instructions `IF_ICMPEQ`, `IF_ICMPNE`, `IF_ICMPLT`, `IF_ICMPLE`, `IF_ICMPGT`, and `IF_ICMPGE` pops two integer values off the operand stack and compares them. It branches if the first value is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to the second value (the one that was at the very top of the stack), respectively. Each instruction has a statement label as its one explicit operand. For example:

```
if_icmpgt  L054
```

The `FCMPG` instruction pops two floating-point values off the operand stack and compares them. If the first value is equal to the second value, it pushes 0 onto the stack. If the first value is less than the second value, it pushes -1, or if the first value is greater than the second value, it pushes 1. The instruction has no explicit operands.

Switch

The Jasmin `LOOKUPSWITCH` instruction is designed for high-level statements such as Java's `switch` statement or Pascal's `CASE` statement. An example:

```
lookupswitch
  -8: L003
   1: L002
   4: L004
   5: L004
   7: L004
  default: L001
```

The `LOOKUPSWITCH` instruction itself has no explicit operands but requires an integer value on the stack which it pops off and compares to the key values listed in the following lines. If the value matches a key value, then the instruction branches to the corresponding label. The key values must be sorted in ascending order, and the last line must be default. The key values of a `LOOKUPSWITCH` instruction must be unique, but the statement labels need not be.[4]

---

[4] Jasmin also has a `TABLESWITCH` instruction which the Pascal compiler will not use in this book.

Call and Return

Jasmin instructions that the Pascal compiler will emit to call a method are `INVOKESTATIC`, `INVOKEVIRTUAL`, and `INVOKENONVIRTUAL`. Each instruction pushes a new stack frame onto the Java runtime stack for the called method.

Instruction `INVOKESTATIC` calls a static method of a Jasmin class. It has one explicit operand, the method signature (fully qualified name and formal parameter type descriptors) of the Jasmin method concatenated with the return type descriptor. If the method

requires any actual parameter values, they are required to be on top of the operand stack, pushed in the correct order. Unless the called method has no return value (i.e., it returns void), the instruction pushes the return value of the method onto the caller's operand stack. Examples:

```
invokestatic  hellomany/sayhello(I)V
invokestatic  hilbert/decompose(I[[F[[F)V
invokestatic  xref/nextchar()C
invokestatic  java/lang/Math/abs(F)F
invokestatic  java/lang/Float.valueOf(F)Ljava/lang/Float;
```

Instruction INVOKEVIRTUAL calls a method on a Jasmin object. It also has one explicit operand, the signature of the Jasmin method concatenated with the return type, and it requires any actual parameter values to be in order at the top of the operand stack. Unless the called method has no return value, the instruction pushes the return value of the method onto the caller's operand stack. Examples:

```
invokevirtual  PascalTextIn.readChar()C
invokevirtual  java/lang/Integer.intValue()I
invokevirtual  java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                              Object;)Ljava/lang/Object;
```

The Pascal compiler will emit the INVOKENONVIRTUAL only to call constructor methods. Jasmin constructor methods are always named <init> and they never return values. Examples:

```
invokenonvirtual  RunTimer/<init>()V
invokenonvirtual  java/util/HashMap/<init>()V
invokenonvirtual  java/util/Scanner.<init>(Ljava/io/InputStream;)V
```

Jasmin has several instructions that return from a method call: RETURN, IRETURN, FRETURN, and ARETURN. Instruction RETURN returns to the caller when the called method does not have a return value. Instructions IRETURN, FRETURN, and ARETURN each pops an integer, floating-point, or address value, respectively, from the operand stack as the return value, and then it returns from a method call. The calling instruction (INVOKESTATIC or INVOKEVIRTUAL) then pushes the return value onto the caller's operand stack. Each of the return instructions pops off the called method's stack frame from the Java runtime stack.

No Operation

Finally, the Jasmin instruction NOP performs no operation. The Pascal compiler will emit this instruction for an empty compound statement. It has no explicit operands.

Remaining Code Generation Methods

You'll examine the remaining code-emitting methods of class CodeGenerator when they are needed in the next three chapters.

Chapter 16

Compiling Programs, Assignment Statements, and Expressions

This is the first of three chapters where you'll generate Jasmin assembly object code for Pascal source programs. You'll be able to assemble the Jasmin code in this chapter and both assemble and execute the Jasmin code in the next two chapters.

Goals and Approach

You'll take a top-down approach to developing the code generator back end of the Pascal compiler. The goals for this chapter include:

- Generate object code for the header, prologue, and epilogue of a Pascal program and of each procedure and function.
- Generate object code for compound statements, simple assignment statements, and expressions.

As mentioned in the previous chapter, the Pascal compiler uses the same front end and intermediate tier that you developed for the Pascal interpreter. The compiler will require only a few changes to its components. You'll test this chapter's compiler by compiling a simple Pascal program.

Compiling Programs

You saw two example Pascal programs compiled into Jasmin assembly object code in the previous chapter. See Listings 15-6a and c and 15-7a and c. The compiler will compile each Pascal program into a Jasmin class, which you'll call the "program class".

Figure 16-1 shows a *code template* for compiling a Pascal source program into Jasmin assembly object code. The template shows the actual Jasmin object code that the code generator emits for the program class.

Figure 16-1: Code template for compiling a Pascal source program

```
.class public program-name          Program header
.super java/lang/Object
```

Code for fields

```
.method public <init>()V            Class constructor

    aload_0
    invokenonvirtual java/lang/Object/<init>()V
    return

.limit locals 1
.limit stack 1
.end method
```

Code for methods

Code for the main method

Design Note

Program Header

The program header consists of the `.class` directive and the `.super` directive. The `.class` directive includes the keyword `public` followed by the Pascal program name in lower case. The `.super` directive specifies the superclass, which for program class is always `java.lang.object`. For example, in Figure 15-7c, you saw that the program header generated for the Pascal source program `HelloMany.pas` is

```
.class public hellomany
.super java/lang/Object
```

Class Constructor

The Pascal compiler always generates the same default constructor for the program class. All Jasmin constructors are named `<init>`. When the constructor executes, slot 0 of the local variables array contains the address of the class instance (i.e., `this` in Java). To create an instance of the program class, the `ALOAD_0` instruction pushes this address onto the operand stack and the `INVOKEVIRTUAL` instruction passes it to the constructor of `java.lang.object`.

Fields

For each Pascal variable declared at nesting level 1, the compiler generates a private static field of the program class. For example,

```
PROGRAM fields;

VAR
    i, j, k : integer;
    x, y    : real;
    p, q    : boolean;
    ch      : char;
    index   : 1..10;
```

compiles to

```
.field private static _runTimer LRunTimer;
.field private static _standardIn LPascalTextIn;

.field private static ch C
.field private static i I
.field private static index I
.field private static j I
.field private static k I
.field private static p Z
.field private static q Z
.field private static x F
.field private static y F
```

The `.field` directive specifies each field of the program class. It uses the `private` and `static` keywords followed by the field name and the type specification. The compiler always generates two "hidden" fields `_runTimer` and `_standardIn`. As you'll see later, the first field keeps track of the amount of time the object program takes to execute and the second field provides runtime input.

Main Method

The compiler generates the main method of the program class when it compiles the main compound statement of the Pascal source program. Figure 16-2 shows the code template.

Figure 16-2: Code template for compiling the main method, i.e., the main compound statement of the Pascal source program

```
.method public static main([Ljava/lang/String;)V
```

```
        new                 RunTimer
        dup
        invokenonvirtual    RunTimer/<init>()V
        putstatic           program-name/_runTimer LRunTimer;
        new                 PascalTextIn
        dup
        invokenonvirtual    PascalTextIn/<init>()V
        putstatic           program-name/_standardIn LPascalTextIn;
```

Code for structured data allocations

Code for compound statement

```
        getstatic           program-name/_runTimer LRunTimer;
        invokevirtual       RunTimer.printElapsedTime()V

        return

.limit locals n
.limit stack m
.end method
```

After the header for the main method, the prologue code creates and initializes a `RunTimer` object and a `PascalTextIn` object. The NEW instruction creates the object and pushes its address onto the operand stack, and then the DUP instruction pushes a copy of the address. The INVOKEVIRTUAL instruction pops off one copy of the address to pass to the class constructor, and the PUTSTATIC instruction stores the other copy into the field. Note that the field name is qualified with the program name.

The main method's epilogue code calls `RunTimer.printElapsedTime()` to compute and print the execution time of the object program.

Classes `RunTime` and `PascalTextIn` are in the *Pascal Runtime Library* that you need to develop. You'll examine this library later in this chapter.
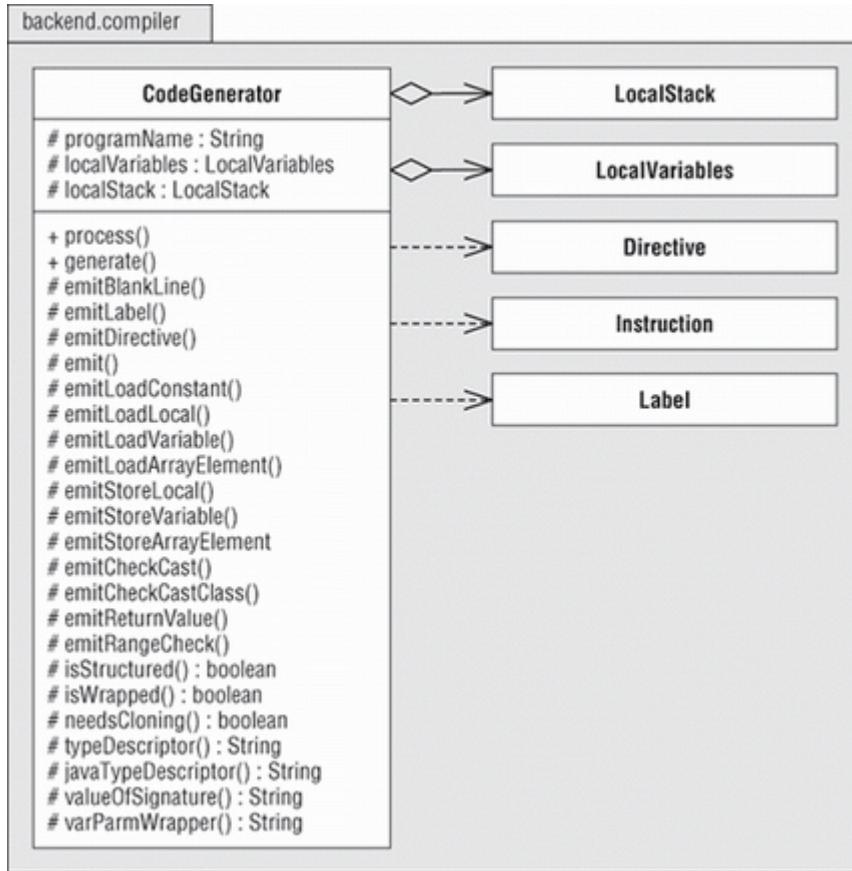
Code Generator Subclasses

Now that you've seen some of the Jasmin assembly object code that the Pascal compiler will generate, you can examine the organization of the compiler's back end code generator.

In the previous chapter, you saw class `CodeGenerator` and most of its code-emitting methods. You also saw classes `Label`, `LocalStack`, and `LocalVariables`, and enumerated types `Directive` and `Instruction`. The UML class diagram in
shows their relationships.
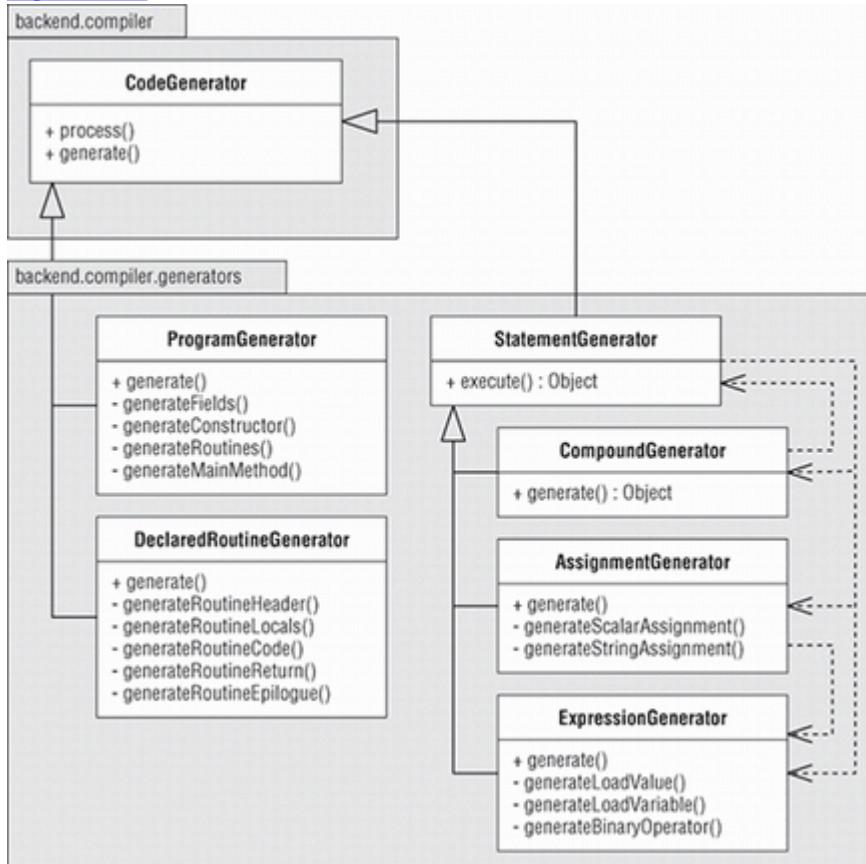
Figure 16-3: Class `CodeGenerator` and its dependencies in the compiler back end

Class `CodeGenerator` is the superclass of the code generator subclasses. The UML class diagram in Figure 16-4 shows some of these subclasses. You'll examine the rest of them in the next two chapters.

Listing 16-1 shows the code generator subclass `ProgramGenerator`. Its methods use the symbol table, parse tree, and the code-emitting methods in its `CodeGenerator` superclass to generate Jasmin assembly object code for Pascal source program according to the code templates in Figures 16-1 and 16-2.

Figure 16-4: Some of the subclasses of class `CodeGenerator`

Listing 16-1: Class `ProgramGenerator`

```
package wci.backend.compiler.generators;

import java.util.ArrayList;
```

```java
import wci.intermediate.*;
import wci.intermediate.symtabimpl.*;
import wci.backend.compiler.*;

import static wci.intermediate.symtabimpl.SymTabKeyImpl.*;
import static wci.intermediate.symtabimpl.DefinitionImpl.*;
import static wci.backend.compiler.Directive.*;
import static wci.backend.compiler.Instruction.*;

/**
 * <h1>ProgramGenerator</h1>
 *
 * <p>Generate code for the main program.</p>
 */
public class ProgramGenerator extends CodeGenerator
{
    private SymTabEntry programId;
    private String programName;

    /**
     * Constructor.
     * @param the parent generator.
     */
    public ProgramGenerator(CodeGenerator parent)
    {
        super(parent);
    }

    /**
     * Generate code for the main program.
     * @param node the root node of the program.
     */
    public void generate(ICodeNode node)
        throws PascalCompilerException
    {
        SymTabEntry programId = symTabStack.getProgramId();

        this.programId = programId;
        this.programName = programId.getName();
        localVariables = new LocalVariables(0);
        localStack = new LocalStack();

        emitDirective(CLASS_PUBLIC, programName);
        emitDirective(SUPER, "java/lang/Object");

        generateFields();
        generateConstructor();
        generateRoutines();
        generateMainMethod();
    }

    /**
     * Generate directives for the fields.
     */
    private void generateFields()
    {
        // Runtime timer and standard in.
        emitBlankLine();
        emitDirective(FIELD_PRIVATE_STATIC, "_runTimer",   "LRunTimer;");
        emitDirective(FIELD_PRIVATE_STATIC, "_standardIn", "LPascalTextIn;");

        SymTab symTab = (SymTab) programId.getAttribute(ROUTINE_SYMTAB);
        ArrayList<SymTabEntry> ids = symTab.sortedEntries();

        emitBlankLine();

        // Loop over all the program&apos;s identifiers and
        // emit a .field directive for each variable.
        for (SymTabEntry id : ids) {
            Definition defn = id.getDefinition();

            if (defn == VARIABLE) {
                emitDirective(FIELD_PRIVATE_STATIC, id.getName(),
                              typeDescriptor(id));
            }
        }
    }

    /**
     * Generate code for the main program constructor.
     */
    private void generateConstructor()
    {
        emitBlankLine();
        emitDirective(METHOD_PUBLIC, "<init>()V");

        emitBlankLine();
        emit(ALOAD_0);
        emit(INVOKENONVIRTUAL, "java/lang/Object/<init>()V");
        emit(RETURN);

        emitBlankLine();
        emitDirective(LIMIT_LOCALS, 1);
        emitDirective(LIMIT_STACK , 1);

        emitDirective(END_METHOD);
    }

    /**
     * Generate code for any nested procedures and functions.
     */
    private void generateRoutines()
        throws PascalCompilerException
    {
        DeclaredRoutineGenerator declaredRoutineGenerator =
            new DeclaredRoutineGenerator(this);
        ArrayList<SymTabEntry> routineIds =
            (ArrayList<SymTabEntry>) programId.getAttribute(ROUTINE_ROUTINES);

        // Generate code for each procedure or function.
        for (SymTabEntry id : routineIds) {
            declaredRoutineGenerator.generate(id);
        }
    }

    /**
     * Generate code for the program body as the main method.
     */
    private void generateMainMethod()
        throws PascalCompilerException
    {
```

```
            emitBlankLine();
            emitDirective(METHOD_PUBLIC_STATIC, "main([Ljava/lang/String;)V");

            generateMainMethodPrologue();

            // Generate code to allocate any arrays, records, and strings.
            StructuredDataGenerator structuredDataGenerator =
                                new StructuredDataGenerator(this);
            structuredDataGenerator.generate(programId);

            generateMainMethodCode();
            generateMainMethodEpilogue();
        }

        /**
         * Generate the main method prologue.
         */
        private void generateMainMethodPrologue()
        {
            String programName = programId.getName();

            // Runtime timer.
            emitBlankLine();
            emit(NEW, "RunTimer");
            emit(DUP);
            emit(INVOKENONVIRTUAL, "RunTimer/<init>()V");
            emit(PUTSTATIC, programName + "/_runTimer", "LRunTimer;");

            // Standard in.
            emit(NEW, "PascalTextIn");
            emit(DUP);
            emit(INVOKENONVIRTUAL, "PascalTextIn/<init>()V");
            emit(PUTSTATIC, programName + "/_standardIn LPascalTextIn;");

            localStack.use(3);
        }

        /**
         * Generate code for the main method.
         */
        private void generateMainMethodCode()
            throws PascalCompilerException
        {
            ICode iCode = (ICode) programId.getAttribute(ROUTINE_ICODE);
            ICodeNode root = iCode.getRoot();

            emitBlankLine();

            // Generate code for the compound statement.
            StatementGenerator statementGenerator = new StatementGenerator(this);
            statementGenerator.generate(root);
        }

        /**
         * Generate the main method epilogue.
         */
        private void generateMainMethodEpilogue()
        {
            // Print the execution time.
            emitBlankLine();
            emit(GETSTATIC, programName + "/_runTimer", "LRunTimer;");
            emit(INVOKEVIRTUAL, "RunTimer.printElapsedTime()V");

            localStack.use(1);

            emitBlankLine();
            emit(RETURN);
            emitBlankLine();

            emitDirective(LIMIT_LOCALS, localVariables.count());
            emitDirective(LIMIT_STACK,  localStack.capacity());
            emitDirective(END_METHOD);
        }
    }
}
```

The `generate()` method creates and initializes the `LocalVariables` and `LocalStack` objects that the code generator will use for the main method. After emitting the program header, it calls the methods that generate code for the program class fields, the class constructor, any procedures and functions, allocations for any structured data, and the main method. It calls `structuredDataGenerator.generate()` to generate code to allocate any structured data (strings, arrays, and records).

Method `generateRoutines()` calls `declaredRoutineGenerator.generate()` to generate the object code for each procedure and function. Method `generateMainMethodCode()` calls `statementGenerator.generate()` to generate object code for the main method's statement, i.e., for the source Pascal program's main compound statement.

Methods `generateMainMethodPrologue()` and `generateMainMethodEpilogue()` each calls `localStack.use()` to keep track of how the instructions they generate will use the operand stack at run time. When `generateMainMethodCode()` creates a `StatementGenerator` object, the statement generator inherits the `LocalVariables` object to keep track of any temporary local variables created at run time. Finally, method `generateMainMethodEpilogue()` calls `localVariables.count()` when it emits the `.limit locals` directive, and it calls `localStack.capacity()` when it emits the `.limit stack` directive.

Listing 16-2 shows a placeholder for the `generate()` method of class `StructuredDataGenerator`. You'll develop this class when you compile string variables in the next chapter, and further again when you compile arrays and records in Chapter 18.

Listing 16-2: Placeholder method `generate()` of class `StructuredDataGenerator`

```
        /**
         * Generate code to allocate the structured data of a program,
         * procedure, or function.
         * @param routineId the routine&apos;s symbol table entry.
         */
        public void generate(SymTabEntry routineId)
        {
        }
```

You'll examine the code generation subclasses `DeclaredRoutineGenerator` and `StatementGenerator` below.

Compiling Procedures and Functions

[Figure 16-5](#) shows the code template for compiling a declared Pascal procedure or function into a private static Jasmin method.

The signature of the procedure or function consists of the routine's name in lower case and the type descriptors of the formal parameters. If the routine is a procedure, the return type descriptor is always v for void. For a function, it's the descriptor of the return type.

For example, the Pascal statements
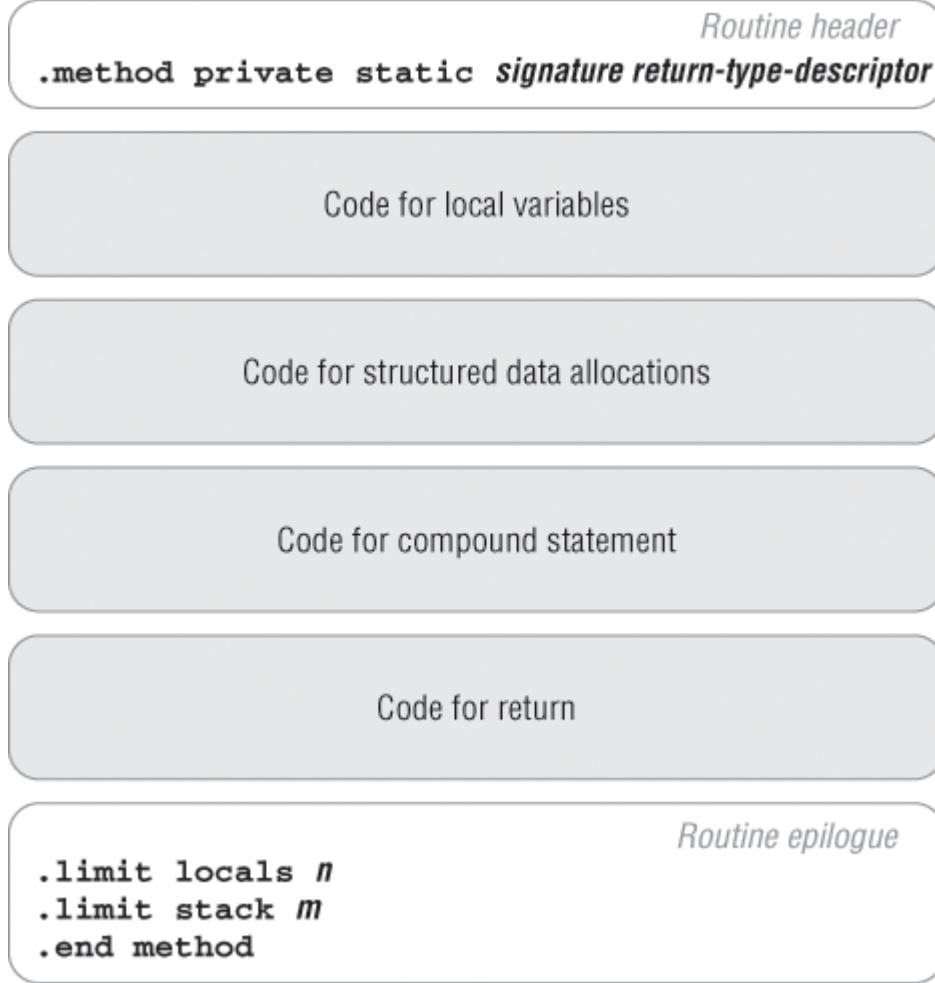
```
TYPE
    arr = ARRAY [1..5] OF real;

FUNCTION func(i, j : integer; x, y : real; p : boolean; ch : char;
             vector : arr; length : integer) : real;
```

compile to

```
.method private static func(IIFFZC[FI)F
```

Note that there is no space between the method signature and its return type descriptor.

Figure 16-5: Code template for compiling a Pascal procedure or function

In Jasmin method for a declared Pascal procedure or function, the values of the method's formal parameters and local variables are stored in slots of the local variables array, as described in the previous chapter. Slot 0 is for the first parameter, and the following parameters and local variables are assigned successive slots. The `.var` directive specifies the slot number and data type for each parameter and local variable. For example, the statements

```
TYPE
    arr = ARRAY [1..5] OF real;

FUNCTION func(i, j : integer; x, y : real; p : boolean; ch : char;
             vector : arr; length : integer) : real;

    VAR
        n : integer;
        z : real;
        w : arr;
```

compile to

```
.method private static func(IIFFZC[FI)F

.var 5 is ch C
.var 0 is i I
.var 1 is j I
.var 7 is length I
.var 8 is n I
.var 4 is p Z
.var 6 is vector [F
.var 10 is w [F
.var 2 is x F
.var 3 is y F
```

```
.var 9 is z F
.var 11 is func F
```

Note that an extra slot, number 11, holds the return value of the function.

Parser and Symbol Table Changes

Modify the front end parser for declared routines to assign slot numbers to each formal parameter and local variable. You also need to make small changes to the symbol table stack and the symbol table to support code generation.

Listing 16-3 shows the two new methods in interface `SymTab` in the `intermediate` package to support code generation.

Listing 16-3: New methods in interface **SymTab**

```
/**
 * @return the next local variables array slot number.
 */
public int nextSlotNumber();

/**
 * @return the maximum local variables array slot number.
 */
public int maxSlotNumber();
```

Listing 16-4 shows the changes to class `SymTabImpl` in package `intermediate.symtabimpl`. There are new fields, the current slot number and the maximum slot number.

Listing 16-4: Changes to class **SymTabImpl**

```
private int slotNumber;        // local variables array slot number
private int maxSlotNumber;     // max slot number value

/**
 * Constructor.
 * @param nestingLevel the nesting level of this entry.
 */
public SymTabImpl(int nestingLevel)
{
    this.nestingLevel = nestingLevel;
    this.slotNumber = -1;
    this.maxSlotNumber = 0;
}

/**
 * @return the next local variables array slot number.
 */
public int nextSlotNumber()
{
    maxSlotNumber = ++slotNumber;
    return slotNumber;
}

/**
 * @return the maximum local variables array slot number.
 */
public int maxSlotNumber()
{
    return maxSlotNumber;
}
```

Listing 16-5 shows the change in the `parse()` method of class `DeclaredRoutineParser` in package `frontend.pascal.parsers`.

Listing 16-5: Changes to method **parse()** of class **DeclaredRoutineParser**

```
// Program: Set the program identifier in the symbol table stack.
// Set the initial local variables array slot number to 1.
if (routineDefn == DefinitionImpl.PROGRAM) {
    symTabStack.setProgramId(routineId);
    symTabStack.getLocalSymTab().nextSlotNumber();  // bump slot number
}
```

For a program, the method sets the first slot number to 1, since slot 0 will be used by the parameter of the main method (the string array of command-line parameter values).

Listing 16-6 shows the change to method `parseIdentifier()` of class `VariableDeclarationsParser` in package `frontend.pascal.parsers`. The method now sets the slot number of a formal parameter or local variable.

Listing 16-6: Change to method **parseIdentifier()** of class **VariableDeclarationsParser**

```
// Enter a new identifier into the symbol table.
if (id == null) {
    id = symTabStack.enterLocal(name);
    id.setDefinition(definition);
    id.appendLineNumber(token.getLineNumber());

    // Set its slot number in the local variables array.
    int slot = id.getSymTab().nextSlotNumber();
    id.setAttribute(SLOT, slot);
}
else {
    errorHandler.flag(token, IDENTIFIER_REDEFINED, this);
}
```

Add two new constants `SLOT` and `WRAP_SLOT` to the enumerated type `SymTabKeyImpl` in package `intermediate.symtabimpl`:

```
// Local variables array slot numbers.
SLOT, WRAP_SLOT
```

You'll see in the next chapter how the code generator uses the wrapper slot number `WRAP_SLOT`.

Generating Code for Procedures and Functions

Listing 16-7 shows the key methods of the code generator subclass `DeclaredRoutineGenerator` that generate code for declared Pascal procedures and functions. This subclass is similar to the subclass `ProgramGenerator` which you examined earlier. It generates the object code specified by the code template in Figure 16-5.

Listing 16-7: Key methods of class `DeclaredRoutineGenerator`

```java
/**
 * Generate code for a declared procedure or function
 * @param routineId the symbol table entry of the routine&apos;s name.
 */
public void generate(SymTabEntry routineId)
    throws PascalCompilerException
{
    this.routineId = routineId;
    this.routineName = routineId.getName();

    SymTab routineSymTab = (SymTab) routineId.getAttribute(ROUTINE_SYMTAB);
    localVariables = new LocalVariables(routineSymTab.maxSlotNumber());
    localStack = new LocalStack();

    // Reserve an extra variable for the function return value.
    if (routineId.getDefinition() == FUNCTION) {
        functionValueSlot = localVariables.reserve();
        routineId.setAttribute(SLOT, functionValueSlot);
    }

    generateRoutineHeader();
    generateRoutineLocals();

    // Generate code to allocate any arrays, records, and strings.
    StructuredDataGenerator structuredDataGenerator =
                            new StructuredDataGenerator(this);
    structuredDataGenerator.generate(routineId);

    generateRoutineCode();
    generateRoutineReturn();
    generateRoutineEpilogue();
}

/**
 * Generate the routine header.
 */
private void generateRoutineHeader()
{
    String routineName = routineId.getName();
    ArrayList<SymTabEntry> parmIds =
        (ArrayList<SymTabEntry>) routineId.getAttribute(ROUTINE_PARMS);
    StringBuilder buffer = new StringBuilder();

    // Procedure or function name.
    buffer.append(routineName);
    buffer.append("(");

    // Parameter and return type descriptors.
    if (parmIds != null) {
        for (SymTabEntry parmId : parmIds) {
            buffer.append(typeDescriptor(parmId));
        }
    }
    buffer.append(")");
    buffer.append(typeDescriptor(routineId));

    emitBlankLine();
    emitDirective(METHOD_PRIVATE_STATIC, buffer.toString());
}

/**
 * Generate directives for the local variables.
 */
private void generateRoutineLocals()
{
    SymTab symTab = (SymTab) routineId.getAttribute(ROUTINE_SYMTAB);
    ArrayList<SymTabEntry> ids = symTab.sortedEntries();

    emitBlankLine();

    // Loop over all the routine&apos;s identifiers and
    // emit a .var directive for each variable and formal parameter.
    for (SymTabEntry id : ids) {
        Definition defn = id.getDefinition();

        if ((defn == VARIABLE) || (defn == VALUE_PARM)
                              || (defn == VAR_PARM)) {
            int slot = (Integer) id.getAttribute(SLOT);
            emitDirective(VAR, slot + " is " + id.getName(),
                        typeDescriptor(id));
        }
    }

    // Emit an extra .var directive for an implied function variable.
    if (routineId.getDefinition() == FUNCTION) {
        emitDirective(VAR, functionValueSlot + " is " + routineName,
                    typeDescriptor(routineId.getTypeSpec()));
    }
}

/**
 * Generate code for the routine&apos;s body.
 */
private void generateRoutineCode()
    throws PascalCompilerException
{
    ICode iCode = (ICode) routineId.getAttribute(ROUTINE_ICODE);
    ICodeNode root = iCode.getRoot();

    emitBlankLine();

    // Generate code for the compound statement.
    StatementGenerator statementGenerator = new StatementGenerator(this);
    statementGenerator.generate(root);
}

/**
 * Generate the routine&apos;s return code.
 */
private void generateRoutineReturn()
```

```
                {
                    emitBlankLine();

                    // Function: Return the value in the implied function variable.
                    if (routineId.getDefinition() == FUNCTION) {
                        TypeSpec type = routineId.getTypeSpec();

                        emitLoadLocal(type, functionValueSlot);
                        emitReturnValue(type);

                        localStack.use(1);
                    }

                    // Procedure: Just return.
                    else {
                        emit(RETURN);
                    }
                }

                /**
                 * Generate the routine&apos;s epilogue.
                 */
                private void generateRoutineEpilogue()
                {
                    emitBlankLine();
                    emitDirective(LIMIT_LOCALS, localVariables.count());
                    emitDirective(LIMIT_STACK,  localStack.capacity());
                    emitDirective(END_METHOD);
                }
```

The `generate()` method creates and initializes the `LocalVariables` and `LocalStack` objects to be inherited by the code generators that will emit the object code for the routine. If the routine is a function, the method calls `localVariables.reserve()` to reserve an extra slot in the local variables array for the return value and sets the `SLOT` attribute of the function name's symbol table entry. It calls methods to generate the routine header and the code for the local variables, respectively. It calls `structuredDataGenerator.generate()` to generate code to allocate any structured data. Finally, it calls methods to generate code for the routine's compound statement, return, and epilogue.

Method `generateRoutineHeader()` emits the `.method private static` directive followed by the routine's signature and return type descriptor. Method `generateRoutineLocals()` generates the `.var` directives for each formal parameter and local variable and, for a function, the return value.

Method `generateRoutineCode()` calls `statementGenerator.generate()` to generate object code for the routine's compound statement. Method `generateRoutineReturn()` emits a simple `RETURN` instruction for a procedure. For a function, it calls `emitLoadLocal()` to emit code to load the return value from its slot in the local variables array and `emitReturnValue()` to emit either `IRETURN`, `FRETURN`, or `ARETURN` depending on the data type.

Finally, method `generateRoutineEpilogue()` calls `localVariables.count()` when it emits the `.limit locals` directive, and it calls `localStack.capacity()` when it emits the `.limit stack` directive.

You'll examine the code generation subclasses `StructuredDataGenerator` and `StatementGenerator` below.

Compiling Assignment Statements and Expressions

In this chapter, the compiler can generate object code for Pascal compound statements, assignment statements, and expressions.

The Statement Code Generator

Listing 16-8 shows the key methods of the code generator subclass `StatementGenerator`.

Listing 16-8: Key methods of class `StatementGenerator`

```
                /**
                 * Generate code for a statement.
                 * To be overridden by the specialized statement executor subclasses.
                 * @param node the root node of the statement.
                 */
                public void generate(ICodeNode node)
                    throws PascalCompilerException
                {
                    ICodeNodeTypeImpl nodeType = (ICodeNodeTypeImpl) node.getType();
                    int line = 0;

                    if (nodeType != COMPOUND) {
                        line = getLineNumber(node);
                        emitDirective(Directive.LINE, line);
                    }

                    // Generate code for a statement according to the type of statement.
                    switch (nodeType) {

                        case COMPOUND: {
                            CompoundGenerator compoundGenerator =
                                            new CompoundGenerator(this);
                            compoundGenerator.generate(node);
                            break;
                        }

                        case ASSIGN: {
                            AssignmentGenerator assignmentGenerator =
                                new AssignmentGenerator(this);
                            assignmentGenerator.generate(node);
                            break;
                        }
                    }

                    // Verify that the stack height after each statement is 0.
                    if (localStack.getSize() != 0) {
                        throw new PascalCompilerException(
                            String.format("Stack size error: size = %d after line %d",
                                    localStack.getSize(), line));
                    }
                }

                /**
```

```
 * Get the source line number of a parse tree node.
 * @param node the parse tree node.
 * @return the line number.
 */
private int getLineNumber(ICodeNode node)
{
    Object lineNumber = null;

    // Go up the parent links to look for a line number.
    while ((node != null) &&
            ((lineNumber = node.getAttribute(ICodeKeyImpl.LINE)) == null)) {
        node = node.getParent();
    }

    return (Integer) lineNumber;
}
```

Method `generate()` calls `compoundGenerator.generate()` and `assignmentGenerator.generate()`. To help ensure that each of these code generation methods generates correct object code that leaves nothing on the operand stack, `generate()` calls `localStack.getSize()` to verify that after executing the generated object code at run time, the stack size is 0. In other words, after executing the code for each statement, nothing should be left on the operand stack.

[Listing 16-9](#) shows class `PascalCompilerException`.

Listing 16-9: Class `PascalCompilerException`

```
package wci.backend.compiler;

/**
 * <h1>PascalCompilerException</h1>
 *
 * <p>Error during the Pascal compiler&apos;s code generation.</p>
 */
public class PascalCompilerException extends Exception
{
    public PascalCompilerException(String message)
    {
        super(message);
    }
}
```

The Compound Statement Code Generator

[Listing 16-10](#) shows the `generate()` method of the statement code generator subclass `CompoundGenerator`.

Listing 16-10: Method `generate()` of class `CompoundGenerator`

```
/**
 * Generate code for a compound statement.
 * @param node the root node of the compound statement.
 */
public void generate(ICodeNode node)
    throws PascalCompilerException
{
    ArrayList<ICodeNode> children = node.getChildren();

    // Loop over the statement children of the COMPOUND node and generate
    // code for each statement. Emit a NOP is there are no statements.
    if (children.size() == 0) {
        emit(NOP);
    }
    else {
        StatementGenerator statementGenerator =
                            new StatementGenerator(this);

        for (ICodeNode child : children) {
            statementGenerator.generate(child);
        }
    }
}
```

For each nested statement in the compound statement, the `generate()` method calls `statementGenerator.generate()`. If there are no nested statements, the method simply emits the `NOP` instruction.

The Assignment Statement Code Generator

In this chapter, the Pascal compiler will generate object code only for assignments to scalar variables. You'll leave string variables, array subscripts, and record fields for the next chapter.

[Listing 16-11](#) shows methods `generate()` and `generateScalarAssignment()` of the statement code generator subclass `AssignmentGenerator`.

Listing 16-11: Methods `generate()` and `generateScalarAssignment()` of class `AssignmentGenerator`

```
/**
 * Generate code for an assignment statement.
 * @param node the root node of the statement.
 */
public void generate(ICodeNode node)
{
    TypeSpec assignmentType = node.getTypeSpec();

    // The ASSIGN node&apos;s children are the target variable
    // and the expression.
    ArrayList<ICodeNode> assignChildren = node.getChildren();
    ICodeNode targetNode = assignChildren.get(0);
    ICodeNode exprNode = assignChildren.get(1);

    SymTabEntry targetId = (SymTabEntry) targetNode.getAttribute(ID);
    TypeSpec targetType = targetNode.getTypeSpec();
    TypeSpec exprType = exprNode.getTypeSpec();
    ExpressionGenerator exprGenerator = new ExpressionGenerator(this);

    int slot;         // local variables array slot number of the target
```

```
            int nestingLevel;  // nesting level of the target
            SymTab symTab;     // symbol table that contains the target id

            // Assign a function value. Use the slot number of the function value.
            if (targetId.getDefinition() == DefinitionImpl.FUNCTION) {
                slot = (Integer) targetId.getAttribute(SLOT);
                nestingLevel = 2;
            }

            // Standard assignment.
            else {
                symTab = targetId.getSymTab();
                slot = (Integer) targetId.getAttribute(SLOT);
                nestingLevel = symTab.getNestingLevel();
            }

            // Generate code to do the assignment.
            generateScalarAssignment(targetType, targetId,
                                     slot, nestingLevel, exprNode, exprType,
                                     exprGenerator);
    }

    /**
     * Generate code to assign a scalar value.
     * @param targetType the data type of the target.
     * @param targetId the symbol table entry of the target variable.
     * @param index the index of the target variable.
     * @param nestingLevel the nesting level of the target variable.
     * @param exprNode the expression tree node.
     * @param exprType the expression data type.
     * @param exprGenerator the expression generator.
     */
    private void generateScalarAssignment(TypeSpec targetType,
                                          SymTabEntry targetId,
                                          int index, int nestingLevel,
                                          ICodeNode exprNode,
                                          TypeSpec exprType,
                                          ExpressionGenerator exprGenerator)
    {
        // Generate code to evaluate the expression.
        // Special cases: float variable := integer constant
        //                float variable := integer expression
        //                char variable  := single-character string constant
        if (targetType == Predefined.realType) {
            if (exprNode.getType() == INTEGER_CONSTANT) {
                int value = (Integer) exprNode.getAttribute(VALUE);
                emitLoadConstant((float) value);
                localStack.increase(1);
            }
            else {
                exprGenerator.generate(exprNode);

                if (exprType == Predefined.integerType) {
                    emit(I2F);
                }
            }
        }
        else if ((targetType == Predefined.charType) &&
                 (exprNode.getType() == STRING_CONSTANT)) {
            int value = ((String) exprNode.getAttribute(VALUE)).charAt(0);
            emitLoadConstant(value);
            localStack.increase(1);
        }
        else {
            exprGenerator.generate(exprNode);
        }

        // Generate code to store the expression value into the target variable.
        emitStoreVariable(targetId, nestingLevel, index);
        localStack.decrease(isWrapped(targetId) ? 2 : 1);
    }
```

Method `generate()` first determines whether or not the assignment is setting the function return value. Then it calls generateScalarAssignment().

Method `generateScalarAssignment()` emits object code to assign scalar values. It emits the `I2F` instruction if necessary to convert an integer value to a floating-point value. The method calls `exprGenerator.generate()` to generate the code that evaluates the expression. However, if the target expression is a constant, the method calls `emitLoadConstant()` to emit the appropriate load constant instruction.[1] Finally, the method calls `emitStoreVariable()` to emit the appropriate store instruction.

---

[1] This is a simple example of a code optimization.

The Expression Code Generator

In this chapter, the Pascal compiler will generate object code for expressions containing only scalar variables. The next two chapters will handle string variables, array subscripts, and record fields.

Listing 16-12 shows the `generate()` method of the statement code generator subclass `ExpressionGenerator`.

Listing 16-12: Method `generate()` of class `ExpressionGenerator`

```
    /**
     * Generate code to evaluate an expression.
     * @param node the root intermediate code node of the compound statement.
     */
    public void generate(ICodeNode node)
    {
        ICodeNodeTypeImpl nodeType = (ICodeNodeTypeImpl) node.getType();
        switch (nodeType) {

            case VARIABLE: {

                // Generate code to load a variable&apos;s value.
                generateLoadValue(node);
                break;
            }

            case INTEGER_CONSTANT: {
                TypeSpec type = node.getTypeSpec();
                Integer value = (Integer) node.getAttribute(VALUE);
```

```
                // Generate code to load a boolean constant
                // 0 (false) or 1 (true).
                if (type == Predefined.booleanType) {
                    emitLoadConstant(value == 1 ? 1 : 0);
                }

                // Generate code to load an integer constant.
                else {
                    emitLoadConstant(value);
                }

                localStack.increase(1);
                break;
            }

            case REAL_CONSTANT: {
                float value = (Float) node.getAttribute(VALUE);

                // Generate code to load a float constant.
                emitLoadConstant(value);

                localStack.increase(1);
                break;
            }

            case STRING_CONSTANT: {
                String value = (String) node.getAttribute(VALUE);

                // Generate code to load a string constant.
                if (node.getTypeSpec() == Predefined.charType) {
                    emitLoadConstant(value.charAt(0));
                }
                else {
                    emitLoadConstant(value);
                }

                localStack.increase(1);
                break;
            }

            case NEGATE: {

                // Get the NEGATE node&apos;s expression node child.
                ArrayList<ICodeNode> children = node.getChildren();
                ICodeNode expressionNode = children.get(0);

                // Generate code to evaluate the expression and
                // negate its value.
                generate(expressionNode);
                emit(expressionNode.getTypeSpec() == Predefined.integerType
                        ? INEG : FNEG);

                break;
            }

            case NOT: {

                // Get the NOT node&apos;s expression node child.
                ArrayList<ICodeNode> children = node.getChildren();
                ICodeNode expressionNode = children.get(0);

                // Generate code to evaluate the expression and NOT its value.
                generate(expressionNode);
                emit(ICONST_1);
                emit(IXOR);

                localStack.use(1);
                break;
            }

            // Must be a binary operator.
            default: generateBinaryOperator(node, nodeType);
        }
    }
```

[Listing 16-13](#) shows methods `generateLoadValue()` and `generateLoadVariable()`. They are very simple in this chapter, but in Chapter 18, they will expand to emit code to handle array subscripts and record fields.

Listing 16-13: Methods `generateLoadValue()` and `generateLoadVariable()` of class `ExpressionGenerator`

```
/**
 * Generate code to load a variable&apos;s value.
 * @param variableNode the variable node.
 */
protected void generateLoadValue(ICodeNode variableNode)
{
    generateLoadVariable(variableNode);
}

/**
 * Generate code to load a variable&apos;s address (structured) or
 * value (scalar).
 * @param variableNode the variable node.
 */
protected TypeSpec generateLoadVariable(ICodeNode variableNode)
{
    SymTabEntry variableId = (SymTabEntry) variableNode.getAttribute(ID);
    TypeSpec variableType = variableId.getTypeSpec();

    emitLoadVariable(variableId);
    localStack.increase(1);

    return variableType;
}
```

[Listing 16-14](#) shows method `generateBinaryOperator()`.

Listing 16-14: Method `generateBinaryOperator()` of class `ExpressionGenerator`

```
/**
 * Generate code to evaluate a binary operator.
 * @param node the root node of the expression.
 * @param nodeType the node type.
 */
private void generateBinaryOperator(ICodeNode node,
                                    ICodeNodeTypeImpl nodeType)
{
```

```java
            // Get the two operand children of the operator node.
            ArrayList<ICodeNode> children = node.getChildren();
            ICodeNode operandNode1 = children.get(0);
            ICodeNode operandNode2 = children.get(1);
            TypeSpec type1 = operandNode1.getTypeSpec();
            TypeSpec type2 = operandNode2.getTypeSpec();

            boolean integerMode = TypeChecker.areBothInteger(type1, type2) ||
                                  (type1.getForm() == ENUMERATION) |
                                  (type2.getForm() == ENUMERATION);
            boolean realMode = TypeChecker.isAtLeastOneReal(type1, type2) ||
                               (nodeType == FLOAT_DIVIDE);
            boolean characterMode = TypeChecker.isChar(type1) &&
                                    TypeChecker.isChar(type2);
            boolean stringMode = type1.isPascalString() &&
                                 type2.isPascalString();

            if (!stringMode) {
                // Emit code to evaluate the first operand.
                generate(operandNode1);
                if (realMode && TypeChecker.isInteger(type1)) {
                    emit(I2F);
                }

                // Emit code to evaluate the second operand.
                generate(operandNode2);
                if (realMode && TypeChecker.isInteger(type2)) {
                    emit(I2F);
                }
            }

            // ====================
            // Arithmetic operators
            // ====================

            if (ARITH_OPS.contains(nodeType)) {
                if (integerMode) {

                    // Integer operations.
                    switch (nodeType) {
                        case ADD:            emit(IADD); break;
                        case SUBTRACT:       emit(ISUB); break;
                        case MULTIPLY:       emit(IMUL); break;
                        case FLOAT_DIVIDE:   emit(FDIV); break;
                        case INTEGER_DIVIDE: emit(IDIV); break;
                        case MOD:            emit(IREM); break;
                    }
                }
                else {

                    // Float operations.
                    switch (nodeType) {
                        case ADD:          emit(FADD); break;
                        case SUBTRACT:     emit(FSUB); break;
                        case MULTIPLY:     emit(FMUL); break;
                        case FLOAT_DIVIDE: emit(FDIV); break;
                    }
                }

                localStack.decrease(1);
            }

            // ==========
            // AND and OR
            // ==========

            else if (nodeType == AND) {
                emit(IAND);
                localStack.decrease(1);
            }
            else if (nodeType == OR) {
                emit(IOR);
                localStack.decrease(1);
            }

            // ====================
            // Relational operators
            // ====================

            else {
                Label trueLabel = Label.newLabel();
                Label nextLabel = Label.newLabel();

                if (integerMode || characterMode) {
                    switch (nodeType) {
                        case EQ: emit(IF_ICMPEQ, trueLabel); break;
                        case NE: emit(IF_ICMPNE, trueLabel); break;
                        case LT: emit(IF_ICMPLT, trueLabel); break;
                        case LE: emit(IF_ICMPLE, trueLabel); break;
                        case GT: emit(IF_ICMPGT, trueLabel); break;
                        case GE: emit(IF_ICMPGE, trueLabel); break;
                    }

                    localStack.decrease(2);
                }

                else if (realMode) {
                    emit(FCMPG);

                    switch (nodeType) {
                        case EQ: emit(IFEQ, trueLabel); break;
                        case NE: emit(IFNE, trueLabel); break;
                        case LT: emit(IFLT, trueLabel); break;
                        case LE: emit(IFLE, trueLabel); break;
                        case GT: emit(IFGT, trueLabel); break;
                        case GE: emit(IFGE, trueLabel); break;
                    }

                    localStack.decrease(2);
                }

                emit(ICONST_0); // false
                emit(GOTO, nextLabel);
                emitLabel(trueLabel);
                emit(ICONST_1); // true
                emitLabel(nextLabel);

                localStack.increase(1);
            }
        }
```
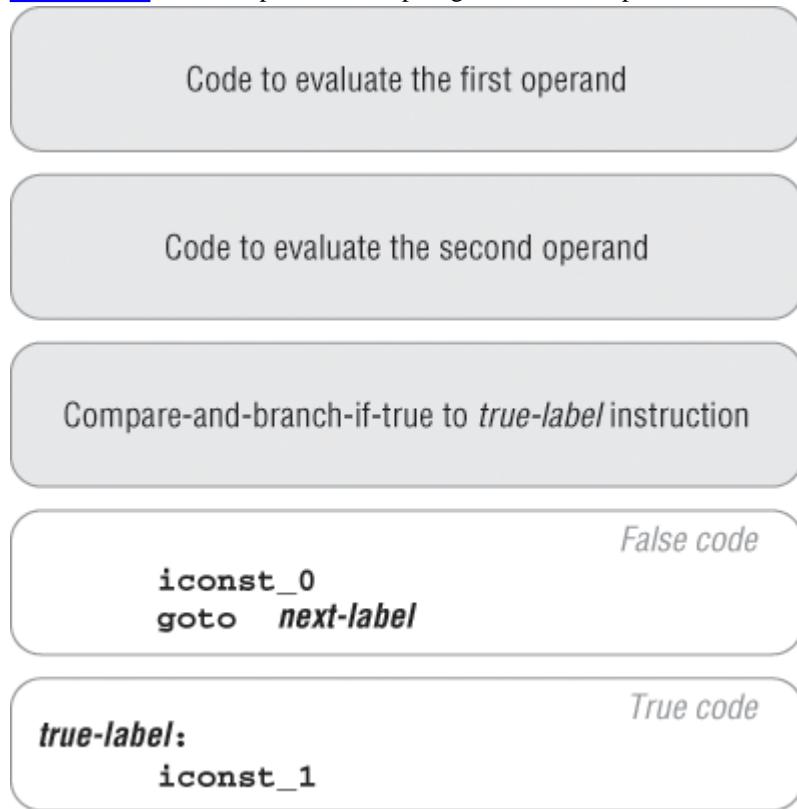
Method `generateBinaryOperator()` first generates code to evaluate the two operands and push their values onto the operand stack. It emits the `I2F` instruction if necessary to convert an integer value to a floating-point value. The method emits code for the arithmetic and the `AND` and `OR` operators in a straightforward manner.

Figure 16-6 shows the code template for a relational expression. The goal of the instruction sequence is to push either the integer value 0 (false) or 1 (true) onto the operand stack.

Figure 16-6: Code template for compiling a relational expression

For integer or character operand values, method `generateBinaryOperator()` emits one of the `IF_ICMPxx` instructions. For floating-point values, the method emits the `FCMPG` instruction followed by one of the `IFxx` instructions. You'll deal with string operand values in the next chapter.

The Pascal Runtime Library

Implement the Pascal Runtime Library as a Java archive (i.e., a `.jar` file). The archive contains several classes that a Pascal source program compiled by our compiler will need to execute at run time.

Range Checking

In class `CodeGenerator`, method `emitStoreVariable()` makes three calls to `emitRangeCheck()`, which is also a method of the class. See Listing 16-15.

Listing 16-15: Method `emitRangeCheck()` of class `CodeGenerator`

```
/**
 * Emit code to perform a runtime range check before an assignment.
 * @param targetType the type of the assignment target.
 */
protected void emitRangeCheck(TypeSpec targetType)
{
    if (targetType.getForm() == SUBRANGE) {
        int min = (Integer) targetType.getAttribute(SUBRANGE_MIN_VALUE);
        int max = (Integer) targetType.getAttribute(SUBRANGE_MAX_VALUE);

        emit(DUP);
        emitLoadConstant(min);
        emitLoadConstant(max);
        emit(INVOKESTATIC, "RangeChecker/check(III)V");

        localStack.use(3);
    }
}
```

Method `emitRangeCheck()` emits code to test the value on top of the operand stack by pushing the minimum and maximum values onto the stack and then calling `RangeChecker/check()`.

Listing 16-16 shows the Pascal Runtime Library class `RangeChecker`.

Listing 16-16: Class `RangeChecker` in the Pascal Runtime Library

```
/**
 * <h1>RangeChecker</h1>
 *
 * <p>Pascal Runtime Library: Perform a runtime range check.</p>
 */
public class RangeChecker
{
    public static void check(int value, int minValue, int maxValue)
        throws PascalRuntimeException
    {
        if ((value < minValue) || (value > maxValue)) {
            throw new PascalRuntimeException(
                String.format("Range error: %1d not in [%1d, %1d]",
                        value, minValue, maxValue));
        }
    }
}
```

Method `check()` checks a value against the minimum and maximum values. If the value falls outside the range, the method throws a new `PascalRuntimeException`. This exception, shown in <u>Listing 16-17</u>, is also in the Pascal Runtime Library.

<u>Listing 16-17:</u> Class `PascalRuntimeException` in the Pascal Runtime Library

```
/**
 * <h1>PascalRuntimeException</h1>
 *
 * <p>Pascal Runtime Library:
 * Exception thrown for an error while executing the generated code.</p>
 */
public class PascalRuntimeException extends Exception
{
    public PascalRuntimeException(String message)
    {
        super(message);
    }
}
```

Pascal Text Input

<u>Listing 16-18</u> shows the placeholder for class `PascalTextIn` in the Pascal Runtime Library. You'll develop this class in the next chapter when you compile the standard Pascal I/O procedures and functions.

<u>Listing 16-18:</u> A placeholder for class `PascalTextIn` in the Pascal Runtime Library

```
/**
 * <h1>PascalTextIn</h1>
 *
 * <p>Pascal Runtime Library:
 * Runtime text input for Pascal programs.</p>
 */
public class PascalTextIn
{
}
```

Building the Library

When you build the Pascal Runtime Library as a Java archive, include all the classes described above.

The Java `jar` utility program creates Java archives. The following command creates the Pascal Runtime Library as the archive `PascalRTL.jar`:

```
jar -cvf PascalRTL.jar PascalTextIn.class RangeChecker.class RunTimer.class \
                PascalRuntimeException.class
```

Program 16-1: Pascal Compiler I

You're ready for an end-to-end test of the Pascal compiler as you've developed it so far in this chapter.

<u>Listing 16-19a</u> shows the Pascal source program `AssignmentTest` which contains some sample assignment statements. Compile it with a command line similar to

```
java -classpath classes Pascal compile AssignmentTest.pas
```

<u>Listing 16-19a:</u> Pascal program `AssignmentTest`

```
001 PROGRAM AssignmentTest;
002
003 VAR
004     tempF, tempC : 0..200;
005     ratio, fahrenheit, centigrade : real;
006     freezing : boolean;
007
008 BEGIN
009     tempF := 72;
010     tempC := 25;
011     ratio := 5.0/9.0;
012
013     fahrenheit := tempF;
014     centigrade := (fahrenheit - 32)*ratio;
015
016     centigrade := tempC;
017     fahrenheit := 32 + centigrade/ratio;
018
019     freezing := fahrenheit < 32
020 END.

            20 source lines.
```

```
                      0 syntax errors.
               0.08 seconds total parsing time.

                     85 instructions generated.
               0.05 seconds total code generation time.
```

Listing 16-19b is the complete listing of the Jasmin assembly object code that the Pascal compiler generates for the source program.

Listing 16-19b: The generated Jasmin object file `assignmenttest.j`

```
.class public assignmenttest
.super java/lang/Object

.field private static _runTimer LRunTimer;
.field private static _standardIn LPascalTextIn;

.field private static centigrade F
.field private static fahrenheit F
.field private static freezing Z
.field private static ratio F
.field private static tempc I
.field private static tempf I

.method public <init>()V

      aload_0
      invokenonvirtual          java/lang/Object/<init>()V
      return

.limit locals 1
.limit stack 1
.end method

.method public static main([Ljava/lang/String;)V

      new      RunTimer
      dup
      invokenonvirtual  RunTimer/<init>()V
      putstatic          assignmenttest/_runTimer LRunTimer;
      new      PascalTextIn
      dup
      invokenonvirtual  PascalTextIn/<init>()V
      putstatic          assignmenttest/_standardIn LPascalTextIn;

.line 9
      bipush  72
      dup
      iconst_0
      sipush  200
      invokestatic      RangeChecker/check(III)V
      putstatic          assignmenttest/tempf I
.line 10
      bipush  25
      dup
      iconst_0
      sipush  200
      invokestatic      RangeChecker/check(III)V
      putstatic          assignmenttest/tempc I
.line 11
      ldc      5.0
      ldc      9.0
      fdiv
      putstatic          assignmenttest/ratio F
.line 13
      getstatic          assignmenttest/tempf I
      i2f
      putstatic          assignmenttest/fahrenheit F
.line 14
      getstatic          assignmenttest/fahrenheit F
      bipush  32
      i2f
      fsub
      getstatic          assignmenttest/ratio F
      fmul
      putstatic          assignmenttest/centigrade F
.line 16
      getstatic          assignmenttest/tempc I
      i2f
      putstatic          assignmenttest/centigrade F
.line 17
      bipush  32
      i2f
      getstatic          assignmenttest/centigrade F
      getstatic          assignmenttest/ratio F
      fdiv
      fadd
      putstatic          assignmenttest/fahrenheit F
.line 19
      getstatic          assignmenttest/fahrenheit F
      bipush  32
      i2f
      fcmpg
      iflt    L001
      iconst_0
      goto    L002
L001:
      iconst_1
L002:
      putstatic          assignmenttest/freezing Z

      getstatic          assignmenttest/_runTimer LRunTimer;
      invokevirtual     RunTimer.printElapsedTime()V

      return

.limit locals 1
.limit stack 4
.end method
```

Because program variables `tempF` and `tempC` have a subrange type, the generated code for the assignment statements in lines 9 and 10 call `RangeChecker.check()`. The generated code for line 19 conforms to the code template for a relational expression shown in .

You'll continue to develop the Pascal compiler in the next chapter, where you'll compile calls to procedures and functions, and string variables and string assignments.

Compiling Procedure and Function Calls and String Operations

This is the second chapter where you'll generate Jasmin assembly object code for Pascal source programs. In this chapter, you'll be able to execute the Jasmin code and produce meaningful output.

Goals and Approach

The goals for this chapter include:
- Generate object code for calls to declared procedures and functions and to the standard procedures and functions, with parameters passed by value and by reference.
- Generate object code to allocate string variables and to perform string assignments.
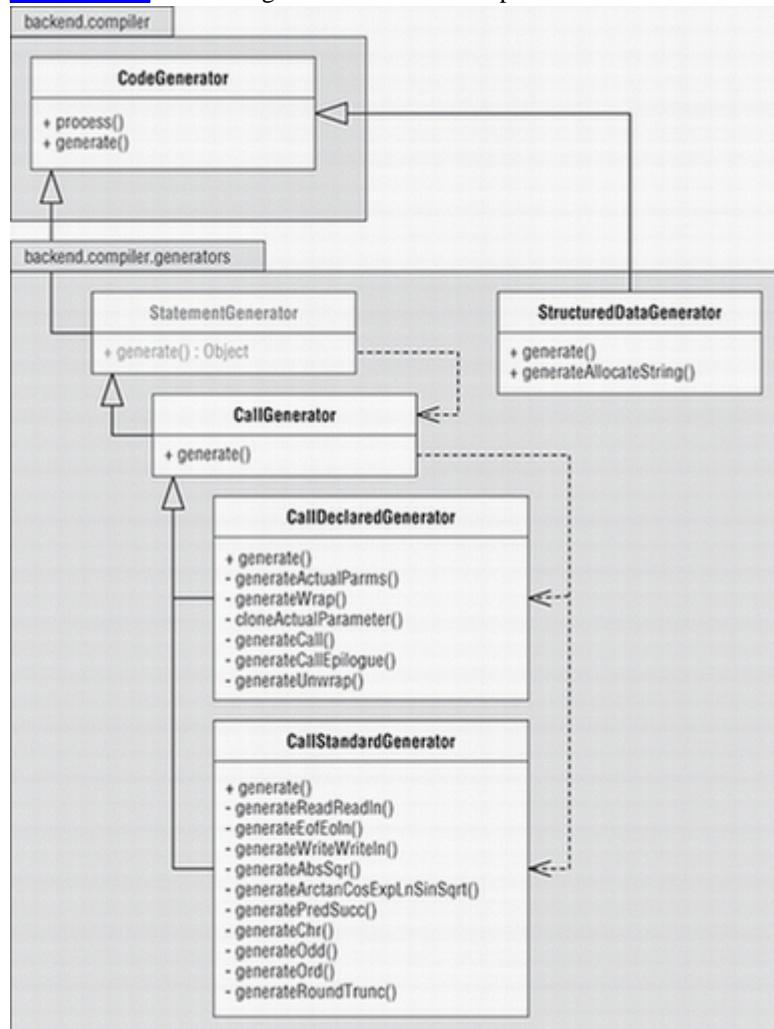
   You'll test this chapter's compiler by compiling and executing a Pascal program that you originally interpreted in Chapter 12. You'll complete the compiler in the next chapter by generating code for the control statements and for arrays and records.

Compiling Procedure and Function Calls

The Pascal compiler, like the Pascal interpreter earlier, must handle calls to declared procedures and functions and to the standard procedures and functions.

   Figure 17-1 expands upon the UML diagram in Figure 16-4 to show the statement code generator subclasses for procedure and function calls.

Figure 17-1: The code generation classes for procedure and function calls

   Listing 17-1 shows the `generate()` method of the statement code generator subclass `CallGenerator`.

Listing 17-1: Method `generate()` of class `CallGenerator`

```
/**
 * Generate code to call a procedure or function.
 * @param node the root node of the call.
 */
public void generate(ICodeNode node)
{
    SymTabEntry routineId = (SymTabEntry) node.getAttribute(ID);
    RoutineCode routineCode =
                (RoutineCode) routineId.getAttribute(ROUTINE_CODE);
    CallGenerator callGenerator = routineCode == DECLARED
                                    ? new CallDeclaredGenerator(this)
                                    : new CallStandardGenerator(this);

    callGenerator.generate(node);
}
```

Class `CallGenerator` is the superclass of classes `CallDeclaredGenerator` and `CallStandardGenerator`.

Listing 17-2 shows the new CALL case in the `generate()` method of class `StatementGenerator` that generates code to call a procedure, and Listing 17-3 shows the new CALL clause in the `generate()` method of class `ExpressionGenerator` that generates code to call a function.

Listing 17-2: The new **CALL** case in method `generate()` of class `StatementGenerator`

```
case CALL: {
    CallGenerator callGenerator = new CallGenerator(this);
    callGenerator.generate(node);
    break;
}
```

Listing 17-3: The new **CALL** case in method `generate()` of class `ExpressionGenerator`

```
case CALL: {

    // Generate code to call a function.
    CallGenerator callGenerator = new CallGenerator(this);
    callGenerator.generate(node);

    break;
}
```

Value Parameters and VAR Parameters

Java (and therefore, Jasmin) passes *all* parameters by value. For a structured parameter value (a string, array, or an object), Java passes (by value) the address of the value.[1] Therefore, passing a structured parameter value is a close approximation of a Pascal VAR parameter, close enough for this book's Pascal compiler.

[1] Passing the address of structured data by value means that the called routine can use that address to modify *components* of that data (e.g., an array element or record field), and those changes will be seen by the caller after the return. However, if the routine changes the value of the address itself (i.e., changes what the address points to), such a change will not be seen by the caller. If Java passed by reference, it would pass the *address of the address* of structured data.

To pass a scalar value as a VAR parameter, first wrap it inside of an object. Therefore, include the wrapper classes `IWrap`, `RWrap`, `BWrap`, and `CWrap` in the Pascal Runtime Library to wrap an integer, real, boolean, or character value, respectively, in order to pass it as a VAR parameter. (How to build the runtime library is described later in this chapter.)

Listing 17-4 shows class `IWrap`. The others are similar.

Listing 17-4: Wrapper class `IWrap` in the Pascal Runtime Library

```
/**
 * <h1>IWrap</h1>
 *
 * <p>Pascal Runtime Library:
 * The wrapper class to pass a scalar integer value by reference.</p>
 * <p>For instructional purposes only.  No warranties.</p>
 */
public class IWrap
{
    public int value;

    public IWrap(int value)
    {
        this.value = value;
    }
}
```

Conversely, to pass a structured parameter value by value, you need to clone the value and pass the address of the clone. Listing 17-5 shows the `Cloner` class in the Pascal Runtime Library.

Listing 17-5: Class `Cloner` in the Pascal Runtime Library

```
/**
 * <h1>Cloner</h1>
 *
 * <p>Pascal Runtime Library:
 * Create a deep clone of an object to pass it by value.</p>
 *
 * <p>Copyright (c) 2009 by Ronald Mak</p>
 * <p>For instructional purposes only.  No warranties.</p>
 */
import java.io.*;

public class Cloner
```

```
{
    public static Object deepClone(Object original)
        throws PascalRuntimeException
    {
        try {
            // Write the original object to a byte array stream.
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(baos);
            oos.writeObject(original);

            // Construct a copy of the original object from the stream.
            ByteArrayInputStream bais =
                new ByteArrayInputStream(baos.toByteArray());
            ObjectInputStream ois = new ObjectInputStream(bais);

            // Return the copy as the deep clone.
            return ois.readObject();
        }
        catch (Exception ex) {
            throw new PascalRuntimeException("Deep clone failed.");
        }
    }
}
```

Method `deepClone()` creates a brute-force deep clone of the original object and returns the clone. It writes the original object's bytes to a byte array stream and then constructs the clone by reading the bytes from byte array stream.[2] You saw exception `PascalRuntimeException` in the Pascal Runtime Library in the previous chapter.

[2] Adapted from http://www.jguru.com/faq/view.jsp?EID=20435.

Listing 17-6 shows the methods in class `CodeGenerator` that support wrapping and cloning.

Listing 17-6: Methods of class `CodeGenerator` that support wrapping and cloning

```
/**
 * Return whether or not a data type is structured.
 * @param type the data type.
 * @return true if the type is a string, array, or record; else false.
 */
protected boolean isStructured(TypeSpec type)
{
    TypeForm form = type.getForm();

    return type.isPascalString() || (form == ARRAY) || (form == RECORD);
}

/**
 * Return whether or not a variable is wrapped to pass by reference.
 * @param variableId the symbol table entry of the variable.
 * @return true if wrapped, false if not.
 */
protected boolean isWrapped(SymTabEntry variableId)
{
    TypeSpec type = variableId.getTypeSpec();
    TypeForm form = type.getForm();
    Definition defn = variableId.getDefinition();

    // Arrays and records are not wrapped.
    return (defn == VAR_PARM) && (form != ARRAY) && (form != RECORD);
}

/**
 * Return whether or not a value needs to be cloned to pass by value.
 * @param formalId the symbol table entry of the formal parameter.
 * @return true if needs wrapping, false if not.
 */
protected boolean needsCloning(SymTabEntry formalId)
{
    TypeSpec type = formalId.getTypeSpec();
    TypeForm form = type.getForm();
    Definition defn = formalId.getDefinition();

    // Arrays and records are normally passed by reference
    // and so must be cloned to be passed by value.
    return (defn == VALUE_PARM) && ((form == ARRAY) || (form == RECORD));
}
```

Calls to Declared Procedures and Functions

Figure 17-2 shows the code template for compiling a call to a declared procedure or function.

Figure 17-2: Code template for compiling a call to a declared procedure or function

Code to evaluate the actual parameters
with any required wrapping and cloning

Call instruction

Code to unwrap any wrapped actual parameters

To call a procedure or function, the Jasmin object code must first evaluate and push the value of each actual parameter, in order, onto the operand stack. If necessary, a parameter value needs to be wrapped or cloned. After the return from the routine, there must be code to unwrap any wrapped parameter values. If the called routine changes the value of a wrapped parameter, the changed value will be unwrapped after the call.

Wrapping Actual Parameter Values

As an example of wrapping and unwrapping VAR parameter values, [Listing 17-7a](#) shows the sample Pascal program WrapTest.

[Listing 17-7a:](#) Pascal program **WrapTest**

```
001 PROGRAM WrapTest;
002
003 VAR
004    i, j : integer;
005
006 PROCEDURE proc(VAR p1 : integer; p2 : integer);
007
008    VAR
009        m, n : integer;
010
011    BEGIN
012        m  := p1;
013        n  := p2;
014        p1 := 10*m;
015        p2 := 10*n;
016    END;
017
018 BEGIN
019    i := 1;
020    j := 2;
021
022    proc(i, j);
023 END.

                   23 source lines.
                    0 syntax errors.
                 0.08 seconds total parsing time.

                   73 instructions generated.
                 0.05 seconds total code generation time.
```

[Listing 17-7b](#) shows the pertinent parts of the Jasmin object code that the Pascal compiler generates.

[Listing 17-7b:](#) Parts of the generated Jasmin object file **wraptest.j**

```
.method private static proc(LIWrap;I)V

.var 2 is m I
.var 3 is n I
.var 0 is p1 LIWrap;
.var 1 is p2 I

.line 12
     aload_0
     getfield     IWrap/value I
     istore_2
.line 13
     iload_1
     istore_3
.line 14
     aload_0
     bipush   10
     iload_2
     imul
     putfield     IWrap/value I
.line 15
     bipush   10
     iload_3
     imul
     istore_1

     return

.limit locals 4
.limit stack 3
.end method
```

```
    .method public static main([Ljava/lang/String;)V

        ...

    .line 19
        iconst_1
        putstatic  wraptest/i I
    .line 20
        iconst_2
        putstatic  wraptest/j I
    .line 22
        new  IWrap
        dup
        getstatic          wraptest/i I
        invokenonvirtual   IWrap/<init>(I)V
        dup
        astore_1
        getstatic          wraptest/j I
        invokestatic       wraptest/proc(LIWrap;I)V
        aload_1
        getfield           IWrap/value I
        putstatic          wraptest/i I

        ...

        return

    .limit locals 2
    .limit stack 3
    .end method
```

In the main method of the Jasmin object code, line 22 is the procedure call.

Because the first formal parameter is a `VAR` parameter, the compiler generates code that creates the wrapper `IWrap` and calls its constructor to wrap the integer value of variable `i`. The `ASTORE_1` instruction stores the address of the wrapper in a newly reserved temporary variable that uses slot 1 of the local variables array and leaves a copy of the address on the operand stack. Since the second formal parameter is passed by value, the generated code simply pushes the value of `j` onto the operand stack.

The `INVOKESTATIC` instruction uses the two parameter values on the operand stack, the address of the wrapper of `i` and the value of `j`, and calls the procedure.

After the call, the wrapped value of `i` needs to be unwrapped. The `ALOAD_1` instruction loads the address of the wrapper, the `GETFIELD` instruction fetches the wrapped value, and the `PUTSTATIC` instruction stores the value into `i`.

In the Jasmin `proc()` method, the `ALOAD_0` and `GETFIELD` instructions for line 12 fetch the wrapped value of the VAR parameter. The `PUTFIELD` instruction for line 14 stores a value into the wrapper.

Call by Reference vs. Call by Value Result

Passing `VAR` parameter values by wrapping them is only an approximation of call by reference. This is actually "call by value result" – the caller wraps the parameter value, the called routine works with and can modify the wrapped value, and upon return, the caller retrieves the possibly modified value from the wrapper. However, if the passed variable is defined in an outer scope and appears in the called routine, the routine would use the unmodified value. In Pascal program `WrapTest`, if the variable `i` appears in a statement in the procedure after line 14, the value used would be the value of `i` as it was at the start of the procedure call, not the value as modified by line 14 via the `VAR` parameter `p1`.

This is another example of a mismatch between the way the Pascal language works and the way the JVM works (the other being the JVM's lack of support for nested procedures and functions). Rather than introduce more complexity in an attempt to simulate true call by reference, the Pascal compiler in this book will stay with call by value result.

Generating Code for Calls

Listing 17-8 shows methods `generate()` and `generateActualParms()` of the call code generator subclass `CallDeclaredGenerator`.

Listing 17-8: Methods `generate()` and `generateActualParms()` of class `CallDeclaredGenerator`

```
    /**
     * Generate code to call to a declared procedure or function.
     * @param node the CALL node.
     */
    public void generate(ICodeNode node)
    {
        // Generate code for any actual parameters.
        if (node.getChildren().size() > 0) {
            generateActualParms(node);
        }

        // Generate code to make the call.
        generateCall(node);

        // Generate code for the epilogue.
        if (node.getChildren().size() > 0) {
            generateCallEpilogue(node);
        }

        // A function call leaves a value on the operand stack.
        SymTabEntry routineId = (SymTabEntry) node.getAttribute(ID);
        if (routineId.getDefinition() == DefinitionImpl.FUNCTION) {
            localStack.increase(1);
        }
    }

    /**
     * Generate code for the actual parameters of a call.
     * @param callNode the CALL parse tree node.
     */
    private void generateActualParms(ICodeNode callNode)
    {
        SymTabEntry routineId = (SymTabEntry) callNode.getAttribute(ID);
        ICodeNode parmsNode = callNode.getChildren().get(0);
        ArrayList<ICodeNode> actualNodes = parmsNode.getChildren();
```

```java
        ArrayList<SymTabEntry> formalIds =
            (ArrayList<SymTabEntry>) routineId.getAttribute(ROUTINE_PARMS);
        ExpressionGenerator exprGenerator = new ExpressionGenerator(this);

        // Iterate over the formal parameters.
        for (int i = 0; i < formalIds.size(); ++i) {
            SymTabEntry formalId = formalIds.get(i);
            ICodeNode actualNode = actualNodes.get(i);
            SymTabEntry actualId = (SymTabEntry) actualNode.getAttribute(ID);
            TypeSpec formalType = formalId.getTypeSpec();
            TypeSpec actualType = actualNode.getTypeSpec();

            // VAR parameter: An actual parameter that is not structured
            //                 needs to be wrapped.
            if (isWrapped(formalId)) {
                Integer wrapSlot = (Integer) actualId.getAttribute(WRAP_SLOT);

                // Already wrapped: Load the wrapper.
                if (wrapSlot != null) {
                    emitLoadLocal(null, wrapSlot);
                    localStack.increase(1);
                }

                // Actual parameter is itself a VAR parameter: No further
                //                 wrapping.
                else if (actualId.getDefinition() == VAR_PARM) {
                    int actualSlot = (Integer) actualId.getAttribute(SLOT);
                    emitLoadLocal(null, actualSlot);
                    localStack.increase(1);
                }

                // Need to wrap: Reserve a temporary variable to hold the
                //                 wrapper&apos;s address.
                else {
                    wrapSlot = localVariables.reserve();
                    actualId.setAttribute(WRAP_SLOT, wrapSlot);
                    generateWrap(actualNode, formalType, wrapSlot,
                                 exprGenerator);
                }
            }

            // Value parameter: Actual parameter is a constant string.
            else if ((formalType == Predefined.charType) &&
                     (actualNode.getType() == STRING_CONSTANT)) {
                int value = ( (String) actualNode.getAttribute(VALUE))
                    .charAt(0);
                emitLoadConstant(value);
                localStack.increase(1);
            }

            // Value parameter: All other types.
            else {
                exprGenerator.generate(actualNode);
                emitRangeCheck(formalType);

                // real formal := integer actual
                if ((formalType == Predefined.realType) &&
                    (actualType.baseType() == Predefined.integerType))
                {
                    emit(I2F);
                }

                // Structured data needs to be cloned.
                else if (needsCloning(formalId)) {
                    cloneActualParameter(formalType);
                }
            }
        }
    }
}
```

If the call has any parameters, method `generate()` calls `generateActualParms()`. It then calls `generateCall()` to generate code for the call itself, and then `generateCallEpilogue()`. If the call is to a function, the method, the call `localStack.increase(1)` accounts for the return value that will be left on top of the operand stack.

Method `generateActualParms()` loops over the formal parameters. For each VAR parameter, it decides whether it needs to emit code to wrap an actual parameter value. It does *not* wrap a scalar value if it already has a wrapper (by having been passed previously as a VAR parameter) or if the value is itself a VAR parameter (and thus already wrapped) being passed again as a VAR parameter. If the method does need to wrap a scalar value, it calls `localVariables.reserve()` to reserve a temporary local variable to store the wrapper's address and calls `generateWrap()`.

If the formal parameter is a value parameter, method `generateActualParms()` emits the appropriate instructions to evaluate and load the actual parameter value. It calls `exprGenerator.generate()` to generate code to evaluate an actual parameter expression. If the parameter value needs to be cloned, the method calls `cloneActualParameter()`.

Listing 17-9 shows methods `generateWrap()` and `cloneActualParameter()`.

Listing 17-9: **Methods `generateWrap()` and `cloneActualParameter()` of class `CallDeclaredGenerator`**

```java
/**
 * Wrap an actual parameter to pass it by reference.
 * in a procedure or function call.
 * @param actualNode the parse tree node of the actual parameter.
 * @param formalType the type specification of the formal parameter
 * @param wrapSlot the slot number of the wrapper
 * @param exprGenerator the expression code generator.
 */
private void generateWrap(ICodeNode actualNode, TypeSpec formalType,
                          int wrapSlot, ExpressionGenerator exprGenerator)
{
    // Wrap the value of an actual parameter.
    String wrapper = varParmWrapper(formalType);  // selected wrapper

    // Create the wrapper.
    emit(NEW, wrapper);
    emit(DUP);
    localStack.increase(2);

    // Generate code to evaluate the actual parameter value.
    exprGenerator.generate(actualNode);

    // Invoke the wrapper&apos;s constructor to wrap the parameter value.
    String init = wrapper + "/<init>(" + typeDescriptor(formalType) + ")V";
    emit(INVOKENONVIRTUAL, init);
    localStack.decrease(1);
```

```
            // Store wrapper&apos;s address into a temporary variable and
            // leave a copy on the operand stack.
            emit(DUP);
            emitStoreLocal(null, wrapSlot);
            localStack.use(1);
    }

    /**
     * Clone an actual parameter value to be passed by value
     * in a procedure or function call.
     * @param formalType the type specification of the formal parameter
     */
    private void cloneActualParameter(TypeSpec formalType)
    {
        emit(INVOKESTATIC, "Cloner.deepClone(Ljava/lang/Object;)" +
                           "Ljava/lang/Object;");
        emitCheckCast(formalType);
    }
```

Method `generateWrap()` emits instructions to create a wrapper, initialize it with the actual parameter value, store the address into the temporary local variable, and leave a copy of the address on the operand stack.

Method `cloneActualParameter()` emits code that calls `Cloner.deepClone()` in the Pascal Runtime Library. It also emits a `CHECKCAST` instruction will verify that the cloned object has the same type as the original object.

[Listing 17-10](#) shows method `generateCall()`.

[Listing 17-10:](#) Method `generateCall()` of class `CallDeclaredGenerator`

```
    /**
     * Generate code to make the call.
     * @param callNode the CALL parse tree node.
     */
    private void generateCall(ICodeNode callNode)
    {
        SymTabEntry routineId = (SymTabEntry) callNode.getAttribute(ID);
        String routineName = routineId.getName();
        ArrayList<SymTabEntry> parmIds =
            (ArrayList<SymTabEntry>) routineId.getAttribute(ROUTINE_PARMS);
        StringBuilder buffer = new StringBuilder();

        // Procedure or function name.
        buffer.append(programName);
        buffer.append("/");
        buffer.append(routineName);
        buffer.append("(");

        // Parameter and return type descriptors.
        if (parmIds != null) {
            for (SymTabEntry parmId : parmIds) {
                buffer.append(typeDescriptor(parmId));
            }
        }
        buffer.append(")");
        buffer.append(typeDescriptor(routineId));

        // Generate a call to the routine.
        emit(INVOKESTATIC, buffer.toString());

        if (parmIds != null) {
            localStack.decrease(parmIds.size());
        }
    }
```

Method `generateCall()` emits the `INVOKESTATIC` instruction, which includes the method signature and return type descriptor. The instruction pops off any actual parameter values on the operand stack, which is accounted for by the call to `localStack.decrease()`.

Finally, [Listing 17-11](#) shows methods `generateCallEpilogue()` and `generateUnwrap()`.

[Listing 17-11:](#) Methods `generateCallEpilogue()` and `generateUnwrap()` of class `CallDeclaredGenerator`

```
    /**
     * Generate code for the call epilogue.
     * @param callNode the CALL parse tree node.
     */
    private void generateCallEpilogue(ICodeNode callNode)
    {
        SymTabEntry routineId = (SymTabEntry) callNode.getAttribute(ID);
        ICodeNode parmsNode = callNode.getChildren().get(0);
        ArrayList<ICodeNode> actualNodes = parmsNode.getChildren();
        ArrayList<SymTabEntry> formalIds =
            (ArrayList<SymTabEntry>) routineId.getAttribute(ROUTINE_PARMS);

        // Iterate over the formal parameters.
        for (int i = 0; i < formalIds.size(); ++i) {
            SymTabEntry formalId = formalIds.get(i);
            ICodeNode actualNode = actualNodes.get(i);
            TypeSpec formalType = formalId.getTypeSpec();

            // Wrapped parameters only.
            if (isWrapped(formalId)) {
                SymTabEntry actualId =
                            (SymTabEntry) actualNode.getAttribute(ID);

                // If the actual parameter is itself a VAR parameter,
                // keep it wrapped. Otherwise unwrap its value.
                if (actualId.getDefinition() != VAR_PARM) {
                    generateUnwrap(actualId, formalType, programName);
                }
            }
        }
    }

    /**
     * Generate the code to unwrap an actual parameter value.
     * @param actualId the symbol table entry of the actual identifier.
     * @param formalType the type specification of the formal parameter
     * @param programName the name of the program.
     */
    private void generateUnwrap(SymTabEntry actualId, TypeSpec formalType,
                                String programName)
    {
        SymTab symTab = actualId.getSymTab();
        int actualSlot = (Integer) actualId.getAttribute(SLOT);
        int wrapSlot = (Integer) actualId.getAttribute(WRAP_SLOT);
        String typeDesc = typeDescriptor(formalType);
```

```
            int nestingLevel = symTab.getNestingLevel();
            String wrapper = varParmWrapper(formalType);  // selected wrapper

            // Load the wrapper and get its value.
            emitLoadLocal(null, wrapSlot);
            emit(GETFIELD, wrapper + "/value", typeDesc);

            // Store the value back into the original variable.
            if (nestingLevel == 1) {
                String actualName = programName + "/" + actualId.getName();
                emit(PUTSTATIC, actualName, typeDesc);
            }
            else {
                emitStoreLocal(formalType, actualSlot);
            }

            localStack.use(1, 2);
    }
```

Method `generateCallEpilogue()` emits the epilogue code shown in the code template in [Figure 17-2](#). It loops over the formal parameters and calls `generateUnwrap()` for each wrapped actual parameter value.

Method `generateUnwrap()` emits the code that unwraps a wrapped value and stores it into the original actual parameter variable, as shown in [Listing 17-8](#).

As an example of the code the Pascal compiler generates for a call to a declared procedure, see [Listing 17-12a](#), which shows the Pascal program `ParmsTest`, and [Listing 17-12b](#), which shows the pertinent parts of the generated Jasmin object code.

[Listing 17-12a:](#) Pascal program `ParmsTest`

```
001 PROGRAM ParmsTest;
002
003 VAR
004     i, j : integer;
005     x, y : real;
006
007 PROCEDURE proc(ii : integer; VAR jj : integer;
008               xx : real; VAR yy : real);
009
010     BEGIN
011         ii := jj;
012         yy := xx;
013     END;
014
015 BEGIN
016     proc(i, j, x, y)
017 END.


            17 source lines.
             0 syntax errors.
          0.09 seconds total parsing time.

            69 instructions generated.
          0.05 seconds total code generation time.
```

[Listing 17-12b:](#) Parts of the generated Jasmin object file `parmstest.j`

```
.method private static proc(ILIWrap;FLRWrap;)V

.var 0 is ii I
.var 1 is jj LIWrap;
.var 2 is xx F
.var 3 is yy LRWrap;

.line 11
    aload_1
    getfield    IWrap/value I
    istore_0
.line 12
    aload_3
    fload_2
    putfield    RWrap/value F

    return

.limit locals 4
.limit stack 2
.end method

.method public static main([Ljava/lang/String;)V

    ...

.line 16
    getstatic        parmstest/i I
    new      IWrap
    dup
    getstatic        parmstest/j I
    invokenonvirtual  IWrap/<init>(I)V
    dup
    astore_1
    getstatic        parmstest/x F
    new      RWrap
    dup
    getstatic        parmstest/y F
    invokenonvirtual  RWrap/<init>(F)V
    dup
    astore_2
    invokestatic      parmstest/proc(ILIWrap;FLRWrap;)V
    aload_1
    getfield    IWrap/value I
    putstatic        parmstest/j I
    aload_2
    getfield RWrap/value F
    putstatic        parmstest/y F

    ...

    return
```

```
        .limit locals 3
        .limit stack 7
        .end method
```

In the generated code for line 11 of procedure `proc()`, the GETFIELD instruction extracts a wrapped value (the VAR parameter `jj`). In line 12, the PUTFIELD instruction sets a wrapped value (the VAR parameter `yy`).

In the main method, the generated code for line 16 first evaluates the actual parameters. The values of `j` and `y` are passed as VAR parameters, and so the code creates wrappers for each. The INVOKESTATIC instruction has a method signature that reflects the fact that the second and fourth parameters are wrapped. After the call instruction, the epilogue code unwraps the values of `j` and `y`.

Calls to the Standard Procedures and Functions

The methods of class `CallStandardGenerator` generate the Jasmin assembly object code to call the standard Pascal procedures and functions. Many of standard Pascal routines can be handled at run time by similar routines from the Java Runtime Library.

Listing 17-13 shows the `generate()` method.

Listing 17-13: Method `generate()` of class `CallStandardGenerator`

```
/**
 * Generate code to call to a standard procedure or function.
 * @param node the CALL node.
 * @return the function value, or null for a procedure call.
 */
public void generate(ICodeNode node)
{
    SymTabEntry routineId = (SymTabEntry) node.getAttribute(ID);
    RoutineCode routineCode =
                    (RoutineCode) routineId.getAttribute(ROUTINE_CODE);
    exprGenerator = new ExpressionGenerator(this);
    ICodeNode actualNode = null;

    // Get the actual parameters of the call.
    if (node.getChildren().size() > 0) {
        ICodeNode parmsNode = node.getChildren().get(0);
        actualNode = parmsNode.getChildren().get(0);
    }

    switch ((RoutineCodeImpl) routineCode) {
        case READ:
        case READLN:  generateReadReadln(node, routineCode); break;

        case WRITE:
        case WRITELN: generateWriteWriteln(node, routineCode); break;

        case EOF:
        case EOLN:    generateEofEoln(node, routineCode); break;

        case ABS:
        case SQR:     generateAbsSqr(routineCode, actualNode); break;

        case ARCTAN:
        case COS:
        case EXP:
        case LN:
        case SIN:
        case SQRT:    generateArctanCosExpLnSinSqrt(routineCode,
                                               actualNode);
                      break;

        case PRED:
        case SUCC:    generatePredSucc(routineCode, actualNode); break;

        case CHR:     generateChr(actualNode); break;
        case ODD:     generateOdd(actualNode); break;
        case ORD:     generateOrd(actualNode); break;

        case ROUND:
        case TRUNC:   generateRoundTrunc(routineCode, actualNode); break;
    }
}
```

read( ), readln( ), eof( ), and eoln( )

Listing 17-14 shows methods `generateReadReadln()` and `generateEofEoln()` that generate code to call the Pascal standard input procedures `read()` and `readln()` and the input functions `eof()` and `eoln()`.

Listing 17-14: Methods `generateReadReadln()` and `generateEofEoln()` of class `CallStandardGenerator`

```
/**
 * Generate code for a call to read or readln.
 * @param callNode the CALL node.
 * @param routineCode the routine code.
 */
private void generateReadReadln(ICodeNode callNode,
                            RoutineCode routineCode)
{
    ICodeNode parmsNode = callNode.getChildren().size() > 0
                            ? callNode.getChildren().get(0)
                            : null;
    String programName = symTabStack.getProgramId().getName();
    String standardInName = programName + "/_standardIn";

    if (parmsNode != null) {
        ArrayList<ICodeNode> actuals = parmsNode.getChildren();

        // Loop to process each actual parameter.
        for (ICodeNode actualNode : actuals) {
            SymTabEntry variableId =
                            (SymTabEntry) actualNode.getAttribute(ID);
            TypeSpec actualType = actualNode.getTypeSpec();
            TypeSpec baseType = actualType.baseType();

            // Generate code to call the appropriate PascalTextIn method.
            emit(GETSTATIC, standardInName, "LPascalTextIn;");
            if (baseType == Predefined.integerType) {
```

```
                emit(INVOKEVIRTUAL, "PascalTextIn.readInteger()I");
            }
            else if (baseType == Predefined.realType) {
                emit(INVOKEVIRTUAL, "PascalTextIn.readReal()F");
            }
            else if (baseType == Predefined.booleanType) {
                emit(INVOKEVIRTUAL, "PascalTextIn.readBoolean()Z");
            }
            else if (baseType == Predefined.charType) {
                emit(INVOKEVIRTUAL, "PascalTextIn.readChar()C");
            }

            localStack.increase(1);

            // Store the value that was read into the actual parameter.
            emitStoreVariable(variableId);

            localStack.decrease(1);
        }
    }

    // READLN: Skip the rest of the input line.
    if (routineCode == READLN) {
        emit(GETSTATIC, standardInName, "LPascalTextIn;");
        emit(INVOKEVIRTUAL, "PascalTextIn.nextLine()V");

        localStack.use(1);
    }
}

/**
 * Generate code for a call to eof or eoln.
 * @param callNode the CALL node.
 * @param routineCode the routine code.
 */
private void generateEofEoln(ICodeNode callNode, RoutineCode routineCode)
{
    String programName = symTabStack.getProgramId().getName();
    String standardInName = programName + "/_standardIn";

    // Generate code to call the appropriate PascalTextIn method.
    emit(GETSTATIC, standardInName, "LPascalTextIn;");
    if (routineCode == EOLN) {
        emit(INVOKEVIRTUAL, "PascalTextIn.atEoln()Z");
    }
    else {
        emit(INVOKEVIRTUAL, "PascalTextIn.atEof()Z");
    }

    localStack.increase(1);
}
```

Both methods `generateReadReadln()` and `generateEofEoln()` emit code to call methods of the `PascalTextIn` class in the Pascal Runtime Library. You'll examine this class in detail shortly.

For example, if `i` is a local integer variable that uses slot 0 of the local variables array, then the Pascal statement

```
readln(i);
```

compiles to the following Jasmin object code, assuming `ReadTest` is the name of the program:

```
getstatic          readtest/_standardIn LPascalTextIn;
invokevirtual      PascalTextIn.readInteger()I
istore_0
getstatic          readtest/_standardIn LPascalTextIn;
invokevirtual      PascalTextIn.nextLine()V
```

write() and writeln()

The Pascal standard procedures `write()` and `writeln()` are more challenging. The compiler generates object code for a Pascal `writeln()` call such as

```
writeln('The square root of ', n:4, ' is ', root:8:4);
```

similarly to the way the Java compiler generates code for the Java statement

```
System.out.println(String.format("The square root of %4d is %8.4f",
                                 n, root));
```

The compiler will generate code that calls the Java methods `System.out.println()` and `String.format()`. Listing 17-15 shows method `generateWriteWriteln()`.

Listing 17-15: Method `generateWriteWriteln()` of class `CallStandardGenerator`

```
/**
 * Generate code for a call to write or writeln.
 * @param callNode the CALL node.
 * @param routineCode the routine code.
 */
private void generateWriteWriteln(ICodeNode callNode,
                                  RoutineCode routineCode)
{
    ICodeNode parmsNode = callNode.getChildren().size() > 0
                            ? callNode.getChildren().get(0)
                            : null;
    StringBuilder buffer = new StringBuilder();
    int exprCount = 0;

    buffer.append("\"");

    // There are actual parameters.
    if (parmsNode != null) {
        ArrayList<ICodeNode> actuals = parmsNode.getChildren();

        // Loop to process each WRITE parameter
        // and build the format string.
        for (ICodeNode writeParmNode : actuals) {
            ArrayList<ICodeNode> children = writeParmNode.getChildren();
            ICodeNode exprNode = children.get(0);
            ICodeNodeType nodeType = exprNode.getType();

            // Append string constants directly to the format string.
            if (nodeType == STRING_CONSTANT) {
                String str = (String)exprNode.getAttribute(VALUE);
                buffer.append(str.replaceAll("%", "%%"));
```

```
            }

            // Create and append the appropriate format specification.
            else {
                TypeSpec dataType = exprNode.getTypeSpec().baseType();
                String typeCode = dataType.isPascalString()          ? "s"
                                  : dataType == Predefined.integerType ? "d"
                                  : dataType == Predefined.realType     ? "f"
                                  : dataType == Predefined.booleanType ? "s"
                                  : dataType == Predefined.charType     ? "c"
                                  :                                       "s";

                ++exprCount;  // count the non-constant string parameters
                buffer.append("%");

                // Process any field width and precision values.
                if (children.size() > 1) {
                    int w = (Integer) children.get(1).getAttribute(VALUE);
                    buffer.append(w == 0 ? 1 : w);
                }
                if (children.size() > 2) {
                    int p = (Integer) children.get(2).getAttribute(VALUE);
                    buffer.append(".");
                    buffer.append(p == 0 ? 1 : p);
                }

                buffer.append(typeCode);
            }
        }

        buffer.append(routineCode == WRITELN ? "\\n\"" : "\"");
    }

    emit(GETSTATIC, "java/lang/System/out", "Ljava/io/PrintStream;");
    localStack.increase(1);

    // WRITELN with no parameters.
    if (parmsNode == null) {
        emit(INVOKEVIRTUAL, "java/io/PrintStream.println()V");
        localStack.decrease(1);
    }

    // WRITE or WRITELN with parameters.
    else {
        ArrayList<ICodeNode> actuals = parmsNode.getChildren();

        // Load the format string.
        emit(LDC, buffer.toString());
        localStack.increase(1);

        // Generate code to create the values array for String.format().
        if (exprCount > 0) {
            emitLoadConstant(exprCount);
            emit(ANEWARRAY, "java/lang/Object");
            localStack.use(3, 1);

            int index = 0;
            ExpressionGenerator exprGenerator =
                                new ExpressionGenerator(this);

            // Loop to generate code to evaluate each actual parameter.
            for (ICodeNode writeParmNode : actuals) {
                ArrayList<ICodeNode> children = writeParmNode.getChildren();
                ICodeNode exprNode = children.get(0);
                ICodeNodeType nodeType = exprNode.getType();
                TypeSpec dataType = exprNode.getTypeSpec().baseType();

                // Skip string constants, which were made part of
                // the format string.
                if (nodeType != STRING_CONSTANT) {
                    emit(DUP);
                    emitLoadConstant(index++);
                    localStack.increase(2);

                    exprGenerator.generate(exprNode);

                    String signature = dataType.getForm() == SCALAR
                        ? valueOfSignature(dataType)
                        : null;

                    // Boolean: Write "true" or "false".
                    if (dataType == Predefined.booleanType) {
                        Label trueLabel = Label.newLabel();
                        Label nextLabel = Label.newLabel();
                        emit(IFNE, trueLabel);
                        emit(LDC, "\"false\"");
                        emit(Instruction.GOTO, nextLabel);
                        emitLabel(trueLabel);
                        emit(LDC, "\"true\"");
                        emitLabel(nextLabel);

                        localStack.use(1);
                    }

                    // Convert a scalar value to an object.
                    if (signature != null) {
                        emit(INVOKESTATIC, signature);
                    }

                    // Store the value into the values vector.
                    emit(AASTORE);
                    localStack.decrease(3);
                }
            }

            // Format the string.
            emit(INVOKESTATIC,
                "java/lang/String/format(Ljava/lang/String;" +
                "[Ljava/lang/Object;)Ljava/lang/String;");
            localStack.decrease(2);
        }

        // Print.
        emit(INVOKEVIRTUAL,
            "java/io/PrintStream.print(Ljava/lang/String;)V");
        localStack.decrease(2);
    }
}
```

A major task for method `generateWriteWriteln()` is to translate the arguments for `writeln()` such as

```
&apos;The square root of &apos;, n:4, &apos; is &apos;, root:8:4
```

into the Java format string

```
"The square root of %4d is %8.4f".
```

If there are actual parameters, the method loops over them. For each parameter that is a string constant such as `'The square root of '` or `' is '`, the method appends it to variable `buffer`, which will hold the Java format string. For each parameter that is not a string constant, the method appends the appropriate format specification to `buffer`, including any field with precision values. For example, for the parameter `root:8:4`, the method appends the format specification `%8.4f` to `buffer`.

The Java `String.format()` method has a variable-length parameter list. The first parameter is the format string. The remaining parameters are the values to be formatted, one for each format specification in the format string. Jasmin passes these remaining parameters as a one-dimensional array of objects. Therefore, if you assume local variables `n` and `root` use slots 0 and 1, respectively, of the local variables array, then the Pascal statement you saw earlier:

```
writeln('The square root of ', n:4, ' is ', root:8:4);
```

compiles to the Jasmin object code:

```
getstatic       java/lang/System/out Ljava/io/PrintStream;
ldc             "The square root of %4d is %8.4f\n"
iconst_2
anewarray       java/lang/Object
dup
iconst_0
iload_0
invokestatic    java/lang/Integer.valueOf(I)Ljava/lang/Integer;
aastore
dup
iconst_1
fload_1
invokestatic    java/lang/Float.valueOf(F)Ljava/lang/Float;
aastore
invokestatic    java/lang/String/format(Ljava/lang/String;[Ljava/lang/
                                  Object;)Ljava/lang/String;
invokevirtual   java/io/PrintStream.print(Ljava/lang/String;)V
```

The GETSTATIC instruction pushes the value of the `out` field of `java.lang.System` onto the operand stack. Its type is `java.io.PrintStream`. The LDC instruction loads the format string.

The ICONST_2 instruction pushes the constant value 2 onto the operand stack that the ANEWARRAY instruction uses to create a one-dimensional array of two elements. The first element, with index value 0 (ICONST_0) is for the value of `n` in slot 0 (ILOAD_0). The INVOKESTATIC call to `java.lang.Integer.valueOf()` converts the scalar value to an `Integer` object. The first AASTORE instruction pops the index value 0 and the object off the operand stack and stores the latter into the first array element. The second element with index value 1 (ICONST_1) is for the value of `root` in slot 1 (FLOAD_1). The INVOKESTATIC call to `java.lang.Float.valueOf()` converts the scalar value to a `Float` object, which the second AASTORE instruction stores into the second array element. The INVOKESTATIC call to `java.lang.String.Format()` creates the formatted string.

What's left on the operand stack after executing all these instructions are the value of `out` and the format string, and they become the parameters for the INVOKEVIRTUAL call to `java.io.PrintStream.print()` method.

Method `generateWriteWriteln()` emits code to convert a false boolean value to the string `"false"` and a true value to the string `"true"`.

Listing 17-16 shows method `valueOfSignature()` of class `CodeGenerator`, which `generateWriteWriteln()` calls to emit the proper `valueOf` method signature and return value.

Listing 17-16: Method `valueOfSignature()` of class `CodeGenerator`

```
/**
 * Return the valueOf() signature for a given scalar type.
 * @param type the scalar type.
 * @return the valueOf() signature.
 */
protected String valueOfSignature(TypeSpec type)
{
    String javaType = javaTypeDescriptor(type);
    String typeCode = typeDescriptor(type);

    return String.format("%s.valueOf(%s)L%s;",
                    javaType, typeCode, javaType);
}
```

abs() and sqr()

Listing 17-17 shows method `generateAbsSqr()` of class `CallStandardGenerator`, which emits code to call the Pascal standard functions `abs()` and `sqr()`.

Listing 17-17: Method `generateAbsSqr()` of class `CallStandardGenerator`

```
/**
 * Generate code for a call to abs or sqr.
 * @param routineCode the routine code.
 * @param actualNode the actual parameter node.
 */
private void generateAbsSqr(RoutineCode routineCode, ICodeNode actualNode)
{
    exprGenerator.generate(actualNode);

    // ABS: Generate code to call the appropriate integer or float
    //      java.lang.Math method.
    // SQR: Multiply the value by itself.
    if (actualNode.getTypeSpec() == Predefined.integerType) {
        if (routineCode == ABS) {
            emit(INVOKESTATIC, "java/lang/Math/abs(I)I");
        }
        else {
            emit(DUP);
            emit(IMUL);
            localStack.use(1);
        }
    }
```

```
        else {
            if (routineCode == ABS) {
                emit(INVOKESTATIC, "java/lang/Math/abs(F)F");
            }
            else {
                emit(DUP);
                emit(FMUL);
                localStack.use(1);
            }
        }
    }
}
```

For `abs()`, method `generateAbsSqr()` emits code to call `java.lang.Math.abs()`, either the version that takes an integer parameter and returns an integer result, or the version that takes a float-point parameter and returns a floating-point result, depending on the type of the parameter. For `sqr()`, the method generates code that multiples the parameter value by itself.

arctan( ), cos( ), exp( ), ln( ), sin( ), and sqrt( )

Listing 17-18 shows method `generateArctanCosExpLnSinSqrt()`, which emits code to call the standard Pascal functions `arctan()`, `cos ()`, `exp()`, `ln()`, `sin()`, and `sqrt()`.

Listing 17-18: Method `generateArctanCosExpLnSinSqrt()` of class `CallStandardGenerator`

```
/**
 * Generate code for a call to arctan, cos, exp, ln, sin, or sqrt.
 * @param routineCode the routine code.
 * @param actualNode the actual parameter node.
 */
private void generateArctanCosExpLnSinSqrt(RoutineCode routineCode,
                                           ICodeNode actualNode)
{
    String function = null;
    exprGenerator.generate(actualNode);

    // Convert an integer or real value to double.
    TypeSpec actualType = actualNode.getTypeSpec();
    if (actualType == Predefined.integerType) {
        emit(I2D);
    }
    else {
        emit(F2D);
    }

    // Select the appropriate java.lang.Math method.
    switch ((RoutineCodeImpl) routineCode) {
        case ARCTAN: function = "atan"; break;
        case COS:    function = "cos";  break;
        case EXP:    function = "exp";  break;
        case SIN:    function = "sin";  break;
        case LN:     function = "log";  break;
        case SQRT:   function = "sqrt"; break;
    }

    // Call the method and convert the result from double to float.
    emit(INVOKESTATIC, "java/lang/Math/" + function + "(D)D");
    emit(D2F);

    localStack.use(1);
}
```

Method `generateArctanCosExpLnSinSqrt()` emits code to call the appropriate `java.lang.Math` method.

The Remaining Standard Functions

Listing 17-19 shows the remaining methods of class `CallStandardGenerator`.

Listing 17-19: The remaining methods of class `CallStandardGenerator`

```
/**
 * Generate code for a call to pred or succ.
 * @param routineCode the routine code.
 * @param actualNode the actual parameter node.
 */
private void generatePredSucc(RoutineCode routineCode, ICodeNode actualNode)
{
    // Generate code to add or subtract 1 from the value.
    exprGenerator.generate(actualNode);
    emit(ICONST_1);
    emit(routineCode == PRED ? ISUB : IADD);

    localStack.use(1);
}

/**
 * Generate code for a call to chr.
 * @param actualNode the actual parameter node.
 */
private void generateChr(ICodeNode actualNode)
{
    exprGenerator.generate(actualNode);
    emit(I2C);
}

/**
 * Generate code for a call to odd.
 * @param actualNode the actual parameter node.
 */
private void generateOdd(ICodeNode actualNode)
{
    // Generate code to leave the rightmost bit of the value
    // on the operand stack.
    exprGenerator.generate(actualNode);
    emit(ICONST_1);
    emit(IAND);

    localStack.use(1);
```

```
            }

            /**
             * Generate code for a call to ord.
             * @param actualNode the actual parameter node.
             */
            private void generateOrd(ICodeNode actualNode)
            {
                // A character value is treated as an integer value.
                if (actualNode.getType() == STRING_CONSTANT) {
                    int value = ((String) actualNode.getAttribute(VALUE)).charAt(0);
                    emitLoadConstant(value);
                    localStack.increase(1);
                }
                else {
                    exprGenerator.generate(actualNode);
                }
            }

            /**
             * Generate code for a call to round or trunc.
             * @param routineCode the routine code.
             * @param actualNode the actual parameter node.
             */
            private void generateRoundTrunc(RoutineCode routineCode,
                                            ICodeNode actualNode)
            {
                exprGenerator.generate(actualNode);

                // ROUND: Generate code to compute floor(value + 0.5).
                if (routineCode == ROUND) {
                    emitLoadConstant(0.5f);
                    emit(FADD);
                    localStack.use(1);
                }

                // Truncate.
                emit(F2I);
            }
```

Method `generatePredSucc()` generates code to call `pred()` and `succ()` by emitting instructions to add or subtract 1 from the integer value on top of the operand stack.

Method `generateChr()` generates code to call `chr()` by simply emitting the `I2C` instruction to convert the integer value on top of the operand stack to character.

Method `generateOdd()` generates code to call `odd()` by emitting the `IAND` instruction to *and* the constant value 1 to the integer value on top of the operand stack. Either a 0 (false) or a 1 (true) will remain on top of the stack.

Method `generateOrd()` generates code to call `ord()` by emitting code to load the character value. At run time, the JVM treats a character value on the operand stack as an integer value.

Method `generateRoundTrunc()` generates code to call `trunc()` simply by emitting the `F2I` instruction to convert a floating-point value on top of the operand stack to integer. For a call to `round()`, the method emits code to add 0.5 to the floating-point value on top of the operand stack before emitting the `F2I` instruction.

The Pascal Runtime Library

Now revisit the Pascal Runtime Library, which you implemented in the previous chapter as the Java archive file `PascalRTL.jar`.

Earlier in this chapter, you saw class `IWrap` (

) and its cousins `RWrap`, `BWrap`, and `CWrap`, and class `Cloner` (), which are all in the library.

Pascal Input Text

You saw in that methods `generateReadReadln()` and `generateEofEoln()` of class `CallStandardGenerator` emit instructions to call methods of class `PascalTextIn`. shows class `PascalTextIn` in the Pascal Runtime Library.

** Class `PascalTextIn` in the Pascal Runtime Library**

```
import java.io.*;

import wci.frontend.Scanner;
import wci.frontend.Source;
import wci.frontend.Token;
import wci.frontend.TokenType;
import wci.frontend.pascal.PascalScanner;

import static wci.frontend.pascal.PascalTokenType.*;

/**
 * <h1>PascalTextIn</h1>
 *
 * <p>Pascal Runtime Library:
 * Runtime text input for Pascal programs, based on the front end scanner.</p>
 */
public class PascalTextIn
{
    private static Scanner scanner;  // based on the Pascal scanner

    static {
        try {
            scanner = new PascalScanner(
                        new Source(
                            new BufferedReader(
                                new InputStreamReader(System.in))));
        }
        catch (Exception ignored) {}
    }

    /**
     * Read the next integer value.
     * @return the integer value.
     * @throws PascalRuntimeException if an error occurred.
     */
    public int readInteger()
```

```java
        throws PascalRuntimeException
{
    Token token = null;

    try {
        token = scanner.nextToken();
        return (Integer) parseNumber(token, true);
    }
    catch (Exception ex) {
        throw new PascalRuntimeException(
                    "Read error: invalid integer value: '" +
                    token.getText() + "'");
    }
}

/**
 * Read the next real value.
 * @return the real value.
 * @throws PascalRuntimeException if an error occurred.
 */
public float readReal()
        throws PascalRuntimeException
{
    Token token = null;

    try {
        token = scanner.nextToken();
        return (Float) parseNumber(token, false);
    }
    catch (Exception ex) {
        throw new PascalRuntimeException(
                    "Read error: invalid real value: '" +
                    token.getText() + "'");
    }
}

/**
 * Read the next boolean value.
 * @return the boolean value.
 * @throws PascalRuntimeException if an error occurred.
 */
public boolean readBoolean()
        throws PascalRuntimeException
{
    Token token = null;

    try {
        token = scanner.nextToken();
        return parseBoolean(token);
    }
    catch (Exception ex) {
        throw new PascalRuntimeException(
                    "Read error: invalid boolean value: '" +
                    token.getText() + "'");
    }
}

/**
 * Read the next character value.
 * @return the character value.
 * @throws PascalRuntimeException if an error occurred.
 */
public char readChar()
        throws PascalRuntimeException
{
    char ch = ' ';

    try {
        ch = scanner.nextChar();

        if ((ch == Source.EOL) || (ch == Source.EOF)) {
            ch = ' ';
        }

        return ch;
    }
    catch (Exception ex) {
        throw new PascalRuntimeException(
                    "Read error: invalid character value: '" +
                    ch + "'");
    }
}

/**
 * Skip the rest of the current input line.
 * @throws PascalRuntimeException if an error occurred.
 */
public void nextLine()
        throws PascalRuntimeException
{
    try {
        scanner.skipToNextLine();
    }
    catch (Exception ex) {}
}

/**
 * Test for the end of the current input line.
 * @return true if at end of line, else false.
 * @throws PascalRuntimeException if an error occurred.
 */
public boolean atEoln()
throws PascalRuntimeException
{
    try {
        return scanner.atEol();
    }
    catch (Exception ex) {
        return false;
    }
}

/**
 * Test for the end of file.
 * @return true if at end of file, else false.
 * @throws PascalRuntimeException if an error occurred.
 */
public boolean atEof()
        throws PascalRuntimeException
{
    try {
        return scanner.atEof();
```

```java
            }
        catch (Exception ex) {
            return false;
        }
    }
}

/**
 * Parse an integer or real value from the standard input.
 * @param token the current input token.
 * @param isInteger true to parse an integer, false to parse a real.
 * @return the integer or real value.
 * @throws Exception if an error occurred.
 */
private Number parseNumber(Token token, boolean isInteger)
    throws Exception
{
    TokenType tokenType = token.getType();
    TokenType sign = null;

    // Leading sign?
    if ((tokenType == PLUS) || (tokenType == MINUS)) {
        sign = tokenType;
        token = scanner.nextToken();
        tokenType = token.getType();
    }

    // Integer value.
    if (tokenType == INTEGER) {
        Number value = sign == MINUS ? -((Integer) token.getValue())
                            : (Integer) token.getValue();
        return isInteger ? value : new Float(((Integer) value).intValue());
    }

    // Real value.
    else if (tokenType == REAL) {
        Number value = sign == MINUS ? -((Float) token.getValue())
                            : (Float) token.getValue();
        return isInteger ? new Integer(((Float) value).intValue()) : value;
    }

    // Bad input.
    else {
        throw new Exception();
    }
}

/**
 * Parse a boolean value from the standard input.
 * @param token the current input token.
 * @param type the input value type.
 * @return the boolean value.
 * @throws Exception if an error occurred.
 */
private Boolean parseBoolean(Token token)
    throws Exception
{
    if (token.getType() == IDENTIFIER) {
        String text = token.getText();

        if (text.equalsIgnoreCase("true")) {
            return new Boolean(true);
        }
        else if (text.equalsIgnoreCase("false")) {
            return new Boolean(false);
        }
        else {
            throw new Exception();
        }
    }
    else {
        throw new Exception();
    }
}
}
```

Class `PascalTextIn` delegates most of its work to the Pascal scanner that you developed for the front end.

Design Note

It is very reasonable to base the Pascal Runtime Library class `PascalTextIn` on the Pascal scanner. The integer, real, boolean, and character constants that appear in Pascal statements have the same syntax as data values that a compiled program can read in at run time.

Methods `readInteger()` and `readReal()` each calls `scanner.nextToken()` and then `parseNumber()`. Method `readBoolean()` calls `scanner.nextToken()` and then `parseBoolean()`. Method `readChar()` calls `scanner.nextChar()` and then converts an end-of-line or end-of-file character to a blank. Each of these methods can throw a `PascalRuntimeException`. Private methods `parseNumber()` and `parseBoolean()` are straightforward parsers of number and boolean values, respectively.

Methods `atEoln()` and `atEof()` call `scanner.atEol()` and `scanner.atEof()`, respectively.

Building the Library

The following command creates this chapter's version of the Pascal Runtime Library:

```
jar -cvf PascalRTL.jar *Wrap.class Cloner.class PaddedString.class \
                PascalRuntimeException.class PascalTextIn.class \
                RangeChecker.class RunTimer.class \
                wci/frontend/EofToken.class wci/frontend/Scanner.class \
                wci/frontend/Source.class wci/frontend/Token.class \
                wci/frontend/TokenType.class \
                wci/frontend/pascal/PascalScanner.class \
                wci/frontend/pascal/PascalToken.class \
                wci/frontend/pascal/PascalTokenType.class \
                wci/frontend/pascal/tokens/*.class wci/message/*.class
```

Because class `PascalTextIn` depends on the Pascal scanner class `PascalScanner`, you also must include all of the latter's dependencies. These include the front end's token classes along with classes that are never used by the compiled code, such as the classes in the `message` package.

You'll examine code generation for Pascal strings and the `PaddedString` class next.

Compiling Strings and String Assignments

Pascal implements a string as an array of characters. Therefore, it is possible to modify characters of a Pascal string simply by setting the corresponding elements of the character array.

A `java.lang.String` object, however, is immutable. Once you've created a Java string, the string value is constant and cannot be changed. Therefore, a Java string is not a good way to implement a Pascal string.

Instead, the Pascal compiler generates code to allocate a `java.lang.StringBuilder` object for each Pascal string. A `StringBuilder` object represents a mutable sequence of characters. Its `setCharAt()` method allows changing the value of any character within the sequence. The `toString()` method returns a `java.lang.String` object for the character sequence.

Allocating String Variables

Earlier in this chapter, you saw calls to `structuredDataGenerator.generate()` in classes `ProgramGenerator` (Listing 17-1) and `DeclaredRoutineGenerator` (Listing 17-8). These two classes delegate to class `StructuredDataGenerator` the work of generating code to allocated structured data, such as arrays, records, and strings. You'll tackle strings in this chapter and arrays and records in the next chapter.

When a Pascal program, procedure, or function declares a local string variable, memory for that variable must be allocated when the program starts, or whenever the procedure or function is called. Pascal allocates memory for local variables on the runtime stack.[3]

---

[3] Dynamic data, created with the standard procedure `new()`, are allocated in the runtime heap, but the pointer variables are allocated on the stack.

The Pascal compiler implements a Pascal string as a `StringBuilder` object. Therefore, when a compiled Pascal program starts, or whenever a procedure or function is called, the generated code must allocate a `StringBuilder` object for each program or local variable that is a string.

Listing 17-21 shows the `generate()` and `generateAllocateString()` methods of class `StructuredDataGenerator`.

Listing 17-21: Methods `generate()` and `generateAllocateString()` of class `StructuredDataGenerator`

```
/**
 * Generate code to allocate the string variables of a program,
 * procedure, or function.
 * @param routineId the routine&apos;s symbol table entry.
 */
public void generate(SymTabEntry routineId)
{
    SymTab symTab = (SymTab) routineId.getAttribute(ROUTINE_SYMTAB);
    ArrayList<SymTabEntry> ids = symTab.sortedEntries();

    // Loop over all the symbol table&apos;s identifiers tp generate
    // data allocation code for string variables.
    emitBlankLine();
    for (SymTabEntry id : ids) {
        if (id.getDefinition() == VARIABLE) {
            TypeSpec idType = id.getTypeSpec();

            if (idType.isPascalString()) {
                generateAllocateString(id, idType);
            }
        }
    }
}

private static final String PADDED_STRING_CREATE =
    "PaddedString.create(I)Ljava/lang/StringBuilder;";

/**
 * Generate code to allocate a string variable as a StringBuilder.
 * @param variableId the symbol table entry of the variable.
 * @param stringType the string data type.
 */
private void generateAllocateString(SymTabEntry variableId,
                                    TypeSpec stringType)
{

    int length = (Integer) stringType.getAttribute(ARRAY_ELEMENT_COUNT);

    // Allocate a blank-filled string of the correct length.
    emitLoadConstant(length);
    emit(INVOKESTATIC, PADDED_STRING_CREATE);
    localStack.increase(1);

    // Store the allocation into the string variable.
    emitStoreVariable(variableId);
    localStack.decrease(1);
}
```

The `generate()` method loops over the symbol table of a program, procedure, or function to look for string variables. For each string variable, it calls `generateAllocateString()`.

Method `generateAllocateString()` first emits a load constant instruction for the string length, which is the number of elements in the Pascal character array. It emits an `INVOKESTATIC` instruction to call `PaddedString.create()`, which takes the length as its parameter and leaves the addresss of a newly created `StringBuilder` object on top of the operand stack. Method `generateAllocateString()` then emits an instruction to store the address into the string variable.

Listing 17-22 shows class `PaddedString`, which is in the Pascal Runtime Library.

Listing 17-22: Class `PaddedString` in the Pascal Runtime Library

```
/**
```

```
 * <h1>PaddedString</h1>
 *
 * <p>Pascal Runtime Library:
 * The implementation of a Pascal string.</p>
 *
 * <p>Copyright (c) 2009 by Ronald Mak</p>
 * <p>For instructional purposes only.  No warranties.</p>
 */
public class PaddedString
{
    public static StringBuilder create(int length)
    {
        return blanks(length, 0);
    }

    public static StringBuilder blanks(int targetLength, int sourceLength)
    {
        int padLength = targetLength - sourceLength;
        StringBuilder padding = new StringBuilder(padLength);

        while (--padLength >= 0) {
            padding.append(&apos; &apos;);
        }

        return padding;
    }
}
```

Method `create()` is a factory method. Given a length, it returns a `StringBuilder` object of that length. It calls `blanks()` to create a StringBuilder object containing blanks.

Method `blanks()` has two parameters, a target length and a source length. It computes the difference between the two lengths. If the target length is greater than the source length, the method creates and initializes a `StringBuilder` object containing the number of blanks equal to the difference. This method will be useful at run time whenever a shorter source string is assigned to a longer string target variable. The shorter string value must be padded with blanks to equal the length of the target string. As you saw with the `create()` method, method `blanks()` can also create an initial string containing blanks.

String Assignments

A Pascal string assignment has three cases:

1. The lengths of the source and target strings are the same. This is a simple assignment.

2. The source string is longer than the target string. Only a substring of the source is assigned to the target; the source string is truncated.

3. The source string is shorter than the target string. After the assignment, the target string needs blank padding.

Suppose `str5`, `strV`, and `str8` are Pascal string variables with lengths 5, 5, and 8, respectively. In the simple case #1, the Pascal compiler generates code for the Pascal assignments

```
str5 := &apos;hello&apos;;
str5 := strV;
```

similar to the code the Java compiler generates for the Java statements

```
str5.setLength(0); str5.append("hello");
strV.setLength(0); strV.append(str5);
```

For case #2, the Pascal assignment

```
str5 := str8;
```

is compiled like the Java statements

```
str5.setLength(0); str5.append(str8.substring(0, 5));
```

For case #3, the Pascal statement

```
str8 := str5;
```

is compiled like the Java statements

```
str8.setLength(0); str8.append(str5).append(PaddedString.blanks(8, 5));
```

Listing 17-23 shows a new version of method `generate()` and a new method `generateStringAssignment()` of class AssignmentExecutor.

Listing 17-23: Method `generateStringAssignment()` of class `AssignmentGenerator`

```
/**
 * Generate code for an assignment statement.
 * @param node the root node of the statement.
 */
public void generate(ICodeNode node)
{
    TypeSpec assignmentType = node.getTypeSpec();

    // The ASSIGN node&apos;s children are the target variable
    // and the expression.
    ArrayList<ICodeNode> assignChildren = node.getChildren();
    ICodeNode targetNode = assignChildren.get(0);
    ICodeNode exprNode = assignChildren.get(1);

    SymTabEntry targetId = (SymTabEntry) targetNode.getAttribute(ID);
    TypeSpec targetType = targetNode.getTypeSpec();
    TypeSpec exprType = exprNode.getTypeSpec();
    ExpressionGenerator exprGenerator = new ExpressionGenerator(this);

    int slot;           // local variables array slot number of the target
    int nestingLevel;  // nesting level of the target
    SymTab symTab;      // symbol table that contains the target id

    // Assign a function value. Use the slot number of the function value.
    if (targetId.getDefinition() == DefinitionImpl.FUNCTION) {
        slot = (Integer) targetId.getAttribute(SLOT);
        nestingLevel = 2;
    }

    // Standard assignment.
```

```java
    else {
        symTab = targetId.getSymTab();
        slot = (Integer) targetId.getAttribute(SLOT);
        nestingLevel = symTab.getNestingLevel();
    }

    // Assign to a VAR parameter: Load the address of the wrapper.
    if (isWrapped(targetId)) {
        emitLoadLocal(null, slot);
        localStack.increase(1);
    }

    // Assign to a Pascal string.
    else if (assignmentType.isPascalString()) {
        emitLoadVariable(targetId);
        localStack.increase(1);
    }

    // Generate code to do the assignment.
    if (targetType.isPascalString()) {
        generateStringAssignment(assignmentType, exprNode,
                                 exprType, exprGenerator);
    }
    else {
        generateScalarAssignment(targetType, targetId,
                                 slot, nestingLevel, exprNode, exprType,
                                 exprGenerator);
    }
}

private static final String SETLENGTH =
    "java/lang/StringBuilder.setLength(I)V";
private static final String PAD_BLANKS =
    "PaddedString.blanks(II)Ljava/lang/StringBuilder;";

private static final String APPEND_STRING =
    "java/lang/StringBuilder.append(Ljava/lang/String;)" +
    "Ljava/lang/StringBuilder;";
private static final String APPEND_CHARSEQUENCE =
    "java/lang/StringBuilder.append(Ljava/lang/CharSequence;)" +
    "Ljava/lang/StringBuilder;";

private static final String STRINGBUILDER_SUBSTRING =
    "java/lang/StringBuilder.substring(II)Ljava/lang/String;";
private static final String STRING_SUBSTRING =
    "java/lang/String.substring(II)Ljava/lang/String;";

/**
 * Generate code to assign a Pascal string value.
 * @param targetType the data type of the target variable.
 * @param exprNode the expression tree node.
 * @param exprType the expression data type.
 * @param exprGenerator the expression code generator.
 */
private void generateStringAssignment(TypeSpec targetType,
                                      ICodeNode exprNode, TypeSpec exprType,
                                      ExpressionGenerator exprGenerator)
{
    int targetLength =
            (Integer) targetType.getAttribute(ARRAY_ELEMENT_COUNT);
    int sourceLength;
    String appender;

    emit(DUP);
    emitLoadConstant(0);
    emit(INVOKEVIRTUAL, SETLENGTH);

    localStack.use(2, 2);

    // Generate code to load the source string.
    if (exprNode.getType() == STRING_CONSTANT) {
        String value = (String) exprNode.getAttribute(VALUE);
        sourceLength = value.length();
        appender = APPEND_STRING;
        emitLoadConstant(value);

        localStack.increase(1);
    }
    else {
        sourceLength = (Integer) exprType.getAttribute(ARRAY_ELEMENT_COUNT);
        appender = APPEND_CHARSEQUENCE;
        exprGenerator.generate(exprNode);
    }

    // Same lengths.
    if (targetLength == sourceLength) {
        emit(INVOKEVIRTUAL, appender);
        localStack.decrease(1);
    }

    // Truncate if necessary.
    else if (targetLength > sourceLength) {
        emitLoadConstant(0);
        emitLoadConstant(targetLength);

        String substringRoutine = exprNode.getType() == STRING_CONSTANT
                                    ? STRING_SUBSTRING
                                    : STRINGBUILDER_SUBSTRING;

        emit(INVOKEVIRTUAL, substringRoutine);
        emit(INVOKEVIRTUAL, appender);

        localStack.use(2, 3);
    }

    // Blank-pad if necessary.
    else {
        emit(INVOKEVIRTUAL, appender);
        emitLoadConstant(targetLength);
        emitLoadConstant(sourceLength);
        emit(INVOKESTATIC,PAD_BLANKS);
        emit(INVOKEVIRTUAL, APPEND_CHARSEQUENCE);

        localStack.use(2, 3);
    }

    emit(POP);
    localStack.decrease(1);
}
```

Method `generateStringAssignment()` first emits code to set the target string length to 0, and then it emits code to load the source string. It selects the appropriate `append()` method of `StringBuilder` depending on whether the source string is a string constant or a `StringBuilder` value. Based on the lengths of the source and target strings, the method emits code to do a simple assignment (case #1), an assignment with truncation (case #2), or an assignment followed by blank padding (case #3).

String Comparisons

[Listing 17-24](#) shows an addittion to method `generateBinaryOperator()` of class `ExpressionGenerator` to generate code for string comparisons.

[Listing 17-24:](#) Code generation for string comparisons by method `generateBinaryOperator()` in class `ExpressionGenerator`

```
else if (stringMode) {

    // Load the value of the first string operand.
    generate(operandNode1);
    if (operandNode1.getType() != STRING_CONSTANT) {
        emit(INVOKEVIRTUAL,
            "java/lang/StringBuilder.toString()" +
            "Ljava/lang/String;");
    }

    // Load the value of the second string operand.
    generate(operandNode2);
    if (operandNode2.getType() != STRING_CONSTANT) {
        emit(INVOKEVIRTUAL,
            "java/lang/StringBuilder.toString()" +
            "Ljava/lang/String;");
    }

    emit(INVOKEVIRTUAL,
        "java/lang/String.compareTo(Ljava/lang/String;)I");

    switch (nodeType) {
        case EQ: emit(IFEQ, trueLabel); break;
        case NE: emit(IFNE, trueLabel); break;
        case LT: emit(IFLT, trueLabel); break;
        case LE: emit(IFLE, trueLabel); break;
        case GT: emit(IFGT, trueLabel); break;
        case GE: emit(IFGE, trueLabel); break;
    }

    localStack.decrease(2);
}
```

The generated code loads each operand of the string comparison. If the operand is a `StringBuilder` value, the code calls the `toString()` method. With two string values on the operand stack, the generated code calls the `java.lang.string.compareTo()` method, which leaves -1, 0, or 1 on top of the operand stack, depending on whether the first string value is less than, equal to, or greater than, respectively, than the second string value. An IF*xx* instruction tests this value, and the rest of the generated code loads either 0 (false) or 1 (true).

[Listing 17-25a](#) shows the Pascal program `StringTest`. The first string parameter of procedure `testStrings()` is a `VAR` parameter and the second string parameter is passed by value.

[Listing 17-25a:](#) Pascal program `StringTest`

```
001 PROGRAM StringTest;
002
003 TYPE
004     string5 = ARRAY [1..5] OF char;
005     string8 = ARRAY [1..8] OF char;
006
007 VAR
008     str5, strV : string5;
009     str8 : string8;
010
011 PROCEDURE testStrings(VAR s5 : string5; s8 : string8);
012
013     VAR
014         b1, b2, b3 : boolean;
015
016     BEGIN
017         b1 := s5 > s8;
018         b2 := s5 < 'goodbye';
019         b3 := 'nobody' >= s8;
020
021         writeln(b1:6, b2:6, b3:6);
022     END;
023
024 BEGIN
025     str5 := 'hello';
026     str8 := 'everyone';
027     writeln('''''', str5, ''', ', str8, '''''');
028
029     testStrings(str5, str8);
030
031     str5 := str8;  {truncate}
032     strV := str5;
033     writeln('''''', str5, ''', ', strV, ''', ', str8, '''''');
034
035     str5 := 'hello';
036     str8 := str5;  {blank pad}
037     writeln('''''', str8, '''''');
038 END.


            38 source lines.
             0 syntax errors.
         0.08 seconds total parsing time.

          216 instructions generated.
```

Listing 17-25b shows the pertinent parts of the generated Jasmin object code.

Listing 17-25b: Parts of the generated Jasmin object file `stringtest.j`

```
.method private static teststrings(Ljava/lang/StringBuilder;Ljava/lang/StringBui
                                                            lder;)V

.var 2 is b1 Z
.var 3 is b2 Z
.var 4 is b3 Z
.var 0 is s5 Ljava/lang/StringBuilder;
.var 1 is s8 Ljava/lang/StringBuilder;

.line 17
      aload_0
      invokevirtual    java/lang/StringBuilder.toString()Ljava/lang/String;
      aload_1
      invokevirtual    java/lang/StringBuilder.toString()Ljava/lang/String;
      invokevirtual    java/lang/String.compareTo(Ljava/lang/String;)I
      ifgt    L001
      iconst_0
      goto    L002
L001:
      iconst_1
L002:
      istore_2
.line 18
      aload_0
      invokevirtual    java/lang/StringBuilder.toString()Ljava/lang/String;
      ldc       "goodbye"
      invokevirtual    java/lang/String.compareTo(Ljava/lang/String;)I
      iflt    L003
      iconst_0
      goto    L004
L003:
      iconst_1
L004:
      istore_3
.line 19
      ldc       "nobody"
      aload_1
      invokevirtual    java/lang/StringBuilder.toString()Ljava/lang/String;
      invokevirtual    java/lang/String.compareTo(Ljava/lang/String;)I
      ifge    L005
      iconst_0
      goto    L006
L005:
      iconst_1
L006:
      istore   4
.line 21
      getstatic        java/lang/System/out Ljava/io/PrintStream;
      ldc         "%6s%6s%6s\n"
      ...
      invokevirtual    java/io/PrintStream.print(Ljava/lang/String;)V

      return

.limit locals 5
.limit stack 8
.end method

.method public static main([Ljava/lang/String;)V

      ...

      iconst_5
      invokestatic     PaddedString.create(I)Ljava/lang/StringBuilder;
      putstatic        stringtest/str5 Ljava/lang/StringBuilder;
      bipush  8
      invokestatic     PaddedString.create(I)Ljava/lang/StringBuilder;
      putstatic        stringtest/str8 Ljava/lang/StringBuilder;
      iconst_5
      invokestatic     PaddedString.create(I)Ljava/lang/StringBuilder;
      putstatic        stringtest/strv Ljava/lang/StringBuilder;

.line 25
      getstatic        stringtest/str5 Ljava/lang/StringBuilder;
      dup
      iconst_0
      invokevirtual    java/lang/StringBuilder.setLength(I)V
      ldc         "hello"
      invokevirtual    java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/l
                                                  ang/StringBuilder;
      pop
.line 26
      getstatic        stringtest/str8 Ljava/lang/StringBuilder;
      dup
      iconst_0
      invokevirtual    java/lang/StringBuilder.setLength(I)V
      ldc         "everyone"
      invokevirtual    java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/l
                                                  ang/StringBuilder;
      pop
.line 27
      getstatic        java/lang/System/out Ljava/io/PrintStream;
      ldc         "&apos;%s, %s&apos;\n"
      ...
      invokevirtual    java/io/PrintStream.print(Ljava/lang/String;)V
.line 29
      getstatic        stringtest/str5 Ljava/lang/StringBuilder;
      getstatic        stringtest/str8 Ljava/lang/StringBuilder;
      invokestatic     Cloner.deepClone(Ljava/lang/Object;)Ljava/lang/Object;
      checkcast        java/lang/StringBuilder
      invokestatic     stringtest/teststrings(Ljava/lang/StringBuilder;Ljava/lan
                                                  g/StringBuilder;)V
.line 31
      getstatic        stringtest/str5 Ljava/lang/StringBuilder;
      dup
      iconst_0
      invokevirtual    java/lang/StringBuilder.setLength(I)V
      getstatic        stringtest/str8 Ljava/lang/StringBuilder;
      iconst_0
      iconst_5
      invokevirtual    java/lang/StringBuilder.substring(II)Ljava/lang/String;
      invokevirtual    java/lang/StringBuilder.append(Ljava/lang/CharSequence;)L
                                                  java/lang/StringBuilder;
      pop
```

```
.line 32
    getstatic       stringtest/strv Ljava/lang/StringBuilder;
    dup
    iconst_0
    invokevirtual   java/lang/StringBuilder.setLength(I)V
    getstatic       stringtest/str5 Ljava/lang/StringBuilder;
    invokevirtual   java/lang/StringBuilder.append(Ljava/lang/CharSequence;)L
                                        java/lang/StringBuilder;
    pop
.line 33
    getstatic       java/lang/System/out Ljava/io/PrintStream;
    ldc             "&apos;%s, %s, %s&apos;\n"
    ...
    invokevirtual   java/io/PrintStream.print(Ljava/lang/String;)V
.line 35
    getstatic       stringtest/str5 Ljava/lang/StringBuilder;
    dup
    iconst_0
    invokevirtual   java/lang/StringBuilder.setLength(I)V
    ldc             "hello"
    invokevirtual   java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/l
                                        ang/StringBuilder;
    pop
.line 36
    getstatic       stringtest/str8 Ljava/lang/StringBuilder;
    dup
    iconst_0
    invokevirtual   java/lang/StringBuilder.setLength(I)V
    getstatic       stringtest/str5 Ljava/lang/StringBuilder;
    invokevirtual   java/lang/StringBuilder.append(Ljava/lang/CharSequence;)L
                                        java/lang/StringBuilder;
    bipush   8
    iconst_5
    invokestatic    PaddedString.blanks(II)Ljava/lang/StringBuilder;
    invokevirtual   java/lang/StringBuilder.append(Ljava/lang/CharSequence;)L
                                        java/lang/StringBuilder;
    pop
.line 37
    getstatic       java/lang/System/out Ljava/io/PrintStream;
    ldc             "&apos;%s&apos;\n"
    ...
    invokevirtual   java/io/PrintStream.print(Ljava/lang/String;)V

    ...

    return

.limit locals 1
.limit stack 7
.end method
```

The generated code for line 29 pushes the actual parameter values onto the operand stack and calls procecedure `testStrings()`. The first parameter is a VAR parameter. The value of the first actual parameter `str5` is already a reference because it is a `StringBuilder` object, and so it simply needs to be loaded. However, to pass the value of the second actual parameter `str8` by value, it needs to be cloned with a call to `Cloner.deepClone()`. The address of the cloned value is left on the operand stack, and the CHECKCAST instruction verifies the value is a `StringBuilder` reference.

The generated code for line 31 shows string truncation with the call to `java.lang.StringBuilder.substring()`, and the generated code for line 36 shows blank padding with the call to `PaddedString.blanks()`. The generated code for lines 17, 18, and 19 show string comparisons with calls to `java.lang.String.compareTo()`.

The output from running the compiled program is

```
&apos;hello, everyone&apos;
 true false  true
&apos;every, every, everyone&apos;
&apos;hello   &apos;
```

Program 17-1: Pascal Compiler II

You're now ready to give this chapter's Pascal compiler an end-to-end test. You'll do this with a Pascal program `Newton1` that has a procedure and a function, calls the standard `read()`, `write()`, and `writeln()` procedures, but only has assignment statements.

A command similar to the following will invoke the Pascal compiler to compile the source file `Newton1.pas`:

```
java -classpath classes Pascal compile Newton1.pas
```

Listing 17-26a shows the printed output.

Listing 17-26a: Printed output from compiling source file `Newton1.pas`

```
001 PROGRAM Newton1;
002
003 CONST
004     epsilon = 1e-6;
005
006 TYPE
007     positive = 0..32767;
008
009 VAR
010     number : positive;
011
012 FUNCTION root(x : real) : real;
013     VAR
014         r : real;
015
016     BEGIN
017         r := 1;
018
019         r := (x/r + r)/2;
020         r := (x/r + r)/2;
021         r := (x/r + r)/2;
022         r := (x/r + r)/2;
023         r := (x/r + r)/2;
024         r := (x/r + r)/2;
025         r := (x/r + r)/2;
026         r := (x/r + r)/2;
```

```
027
028        root := r;
029      END;
030
031 PROCEDURE print(n : integer; root : real);
032      BEGIN
033          writeln(&apos;The square root of &apos;, n:4, &apos; is &apos;, root:8:4);
034      END;
035
036 BEGIN
037     writeln;
038     write(&apos;Enter a new number: &apos;);
039     read(number);
040     print(number, root(number));
041
042     writeln;
043     write(&apos;Enter a new number: &apos;);
044     read(number);
045     print(number, root(number));
046 END.
```

```
         46 source lines.
          0 syntax errors.
       0.09 seconds total parsing time.

        188 instructions generated.
       0.03 seconds total code generation time.
```

The compiler generates the Jasmin object file `newton1.j`, shown in its entirety in .

The generated Jasmin object file `newton1.j`

```
.class public newton1
.super java/lang/Object

.field private static _runTimer LRunTimer;
.field private static _standardIn LPascalTextIn;

.field private static number I

.method public <init>()V

    aload_0
    invokenonvirtual  java/lang/Object/<init>()V
    return

.limit locals 1
.limit stack 1
.end method

.method private static root(F)F

.var 1 is r F
.var 0 is x F
.var 2 is root F

.line 17
    fconst_1
    fstore_1
.line 19
    fload_0
    fload_1
    fdiv
    fload_1
    fadd
    iconst_2
    i2f
    fdiv
    fstore_1
.line 20
    fload_0
    fload_1
    fdiv
    fload_1
    fadd
    iconst_2
    i2f
    fdiv
    fstore_1
.line 21
    fload_0
    fload_1
    fdiv
    fload_1
    fadd
    iconst_2
    i2f
    fdiv
    fstore_1
.line 22
    fload_0
    fload_1
    fdiv
    fload_1
    fadd
    iconst_2
    i2f
    fdiv
    fstore_1
.line 23
    fload_0
    fload_1
    fdiv
    fload_1
    fadd
    iconst_2
    i2f
    fdiv
    fstore_1
.line 24
    fload_0
    fload_1
    fdiv
    fload_1
    fadd
    iconst_2
    i2f
```

```
        fdiv
        fstore_1
.line 25
        fload_0
        fload_1
        fdiv
        fload_1
        fadd
        iconst_2
        i2f
        fdiv
        fstore_1
.line 26
        fload_0
        fload_1
        fdiv
        fload_1
        fadd
        iconst_2
        i2f
        fdiv
        fstore_1
.line 28
        fload_1
        fstore_2

      fload_2
      freturn

.limit locals 3
.limit stack 2
.end method

.method private static print(IF)V

.var 0 is n I
.var 1 is root F

.line 33
        getstatic         java/lang/System/out Ljava/io/PrintStream;
        ldc       "The square root of %4d is %8.4f\n"
        iconst_2
        anewarray         java/lang/Object
        dup
        iconst_0
        iload_0
        invokestatic        java/lang/Integer.valueOf(I)Ljava/lang/Integer;
        aastore
        dup
        iconst_1
        fload_1
        invokestatic        java/lang/Float.valueOf(F)Ljava/lang/Float;
        aastore
        invokestatic        java/lang/String/format(Ljava/lang/String;[Ljava/lang/Obj
                                              ect;)Ljava/lang/String;
        invokevirtual     java/io/PrintStream.print(Ljava/lang/String;)V

        return

.limit locals 2
.limit stack 7
.end method

.method public static main([Ljava/lang/String;)V

        new       RunTimer
        dup
        invokenonvirtual  RunTimer/<init>()V
        putstatic         newton1/_runTimer LRunTimer;
        new       PascalTextIn
        dup
        invokenonvirtual  PascalTextIn/<init>()V
        putstatic         newton1/_standardIn LPascalTextIn;

.line 37
        getstatic         java/lang/System/out Ljava/io/PrintStream;
        invokevirtual     java/io/PrintStream.println()V
.line 38
        getstatic         java/lang/System/out Ljava/io/PrintStream;
        ldc       "Enter a new number: "
        invokevirtual     java/io/PrintStream.print(Ljava/lang/String;)V
.line 39
        getstatic         newton1/_standardIn LPascalTextIn;
        invokevirtual     PascalTextIn.readInteger()I
        dup
        iconst_0
        sipush   32767
        invokestatic      RangeChecker/check(III)V
        putstatic         newton1/number I
.line 40
        getstatic         newton1/number I
        getstatic         newton1/number I
        i2f
        invokestatic      newton1/root(F)F
        invokestatic      newton1/print(IF)V
.line 42
        getstatic         java/lang/System/out Ljava/io/PrintStream;
        invokevirtual     java/io/PrintStream.println()V
.line 43
        getstatic         java/lang/System/out Ljava/io/PrintStream;
        ldc       "Enter a new number: "
        invokevirtual     java/io/PrintStream.print(Ljava/lang/String;)V
.line 44
        getstatic         newton1/_standardIn LPascalTextIn;
        invokevirtual     PascalTextIn.readInteger()I
        dup
        iconst_0
        sipush   32767
        invokestatic      RangeChecker/check(III)V
        putstatic         newton1/number I
.line 45
        getstatic         newton1/number I
        getstatic         newton1/number I
        i2f
        invokestatic      newton1/root(F)F
        invokestatic      newton1/print(IF)V

        getstatic         newton1/_runTimer LRunTimer;
        invokevirtual     RunTimer.printElapsedTime()V

        return
```

```
.limit locals 1
.limit stack 4
.end method
```

If the Jasmin assembler is installed in directory `/jasmin-2.3`, then a command similar to the following will assemble file `newton1.j` and create the file `newton1.class` in the current directory:

```
java -jar /jasmin-2.3/jasmin.jar newton1.j
```

If the Pascal Runtime Library archive file `PascalRTL.jar` is also in the current directory, then a command similar to the following will run the compiled program:

```
java -classpath .:PascalRTL.jar newton1
```

If you enter `2` in response to the first prompt and `-2` to the second prompt, then output from running the compiled program will be

```
Enter a new number: 2
The square root of    2 is   1.4142

Enter a new number: -2
Exception in thread "main" PascalRuntimeException: Range error: -2 not in
                                                  [0, 32767]
        at RangeChecker.check(RangeChecker.java:15)
        at newton1.main(newton1.j:44)
```

Note that the stack trace for the runtime range error points to line 44 of the Pascal source program.

In the following chapter, you'll complete the Pascal compiler by adding the code generators for the Pascal control statements. The completed compiler will also generate code to allocate arrays and records and for variables that have array subscripts and record fields.

Chapter 18

Compiling Control Statements, Arrays, and Records

In this chapter, you'll complete the Pascal compiler by generating code for the control statements and for arrays and records. You'll be able to compile and execute a variety of source programs written in the Pascal subset that you've been using for this book.

Goals and Approach

The goals for this chapter include:
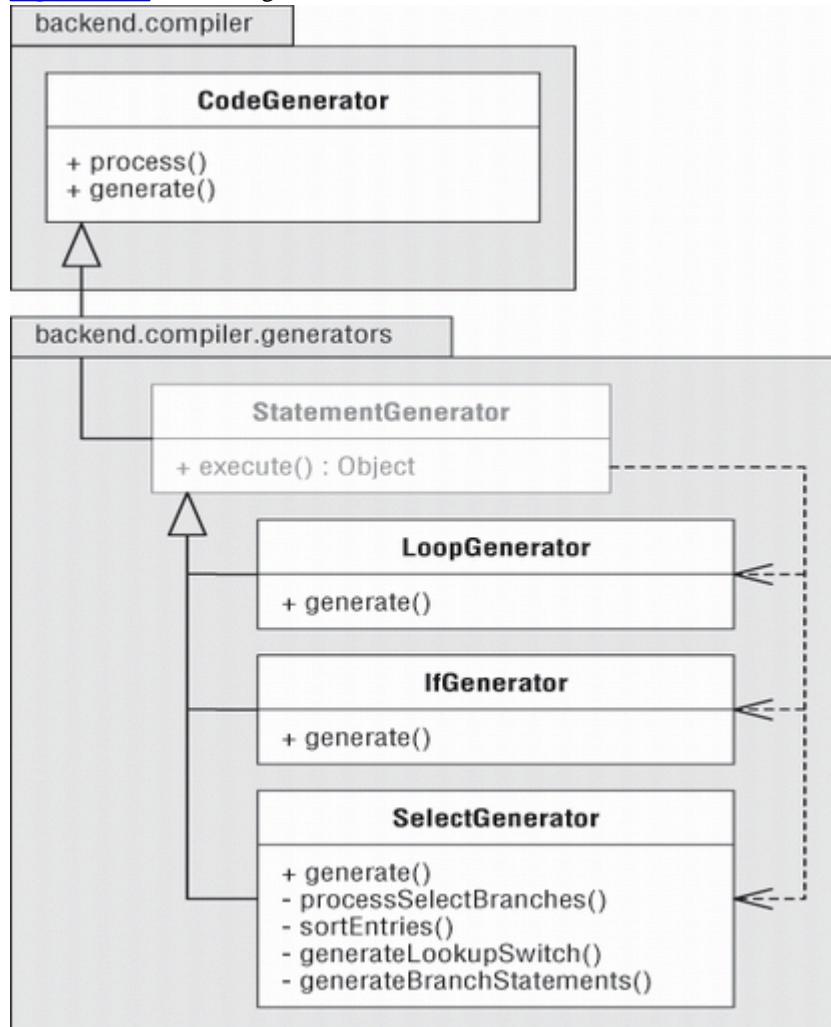
- Generate object code for Pascal's control statements.
- Generate object code to allocate array and record data, and for assignments statements and expressions that include subscripted variables and record fields.

As you did in the previous two chapters, you'll use code templates to guide your code generation, and you'll test the compiler by compiling and executing Pascal programs.

Compiling Control Statements

Figure 18-1 shows the UML class diagrams for the statement code generator subclasses `LoopGenerator`, `IfGenerator`, and `SelectGenerator`.

Figure 18-1: The code generation classes for control statements

Looping Statements

[Figure 18-2](#) shows the code template for compiling looping statements. This general template will work for Pascal's REPEAT, WHILE, and FOR statements.

As you saw in Chapter 16, the generated object code for a boolean expression will leave either the value 0 (false) for 1 (true) on top of the operand stack at run time. Therefore, the IFNE instruction tests whether the value is not equal to 0 (i.e., the value is true). If so, the instruction branches out of the loop.

[Listing 18-1](#) shows method generate() of the statement code generator subclass LoopGenerator.

[Figure 18-2:](#) Code template for compiling looping statements

[Listing 18-1:](#) Method generate() of class LoopGenerator

```
/**
 * Generate code for a looping statement.
 * @param node the root node of the statement.
 */
public void generate(ICodeNode node)
    throws PascalCompilerException
{
    ArrayList<ICodeNode> loopChildren = node.getChildren();
    ExpressionGenerator expressionGenerator = new ExpressionGenerator(this);
    StatementGenerator statementGenerator = new StatementGenerator(this);
    Label loopLabel = Label.newLabel();
    Label nextLabel = Label.newLabel();

    emitLabel(loopLabel);

    // Generate code for the children of the LOOP node.
    for (ICodeNode child : loopChildren) {
        ICodeNodeTypeImpl childType = (ICodeNodeTypeImpl) child.getType();

        // TEST node: Generate code to test the boolean expression.
        if (childType == TEST) {
            ICodeNode expressionNode = child.getChildren().get(0);

            expressionGenerator.generate(expressionNode);
            emit(IFNE, nextLabel);

            localStack.decrease(1);
        }

        // Statement node: Generate code for the statement.
        else {
            statementGenerator.generate(child);
        }
    }

    emit(GOTO, loopLabel);
    emitLabel(nextLabel);
}
```

Method generate() simply loops over the children of the LOOP parse tree node. For a TEST child node, it calls expressionGenerator.generate() to generate code for the boolean expression and then it emits the IFNE instruction. For a statement child node, the method calls statementGenerator.generate() to generate code for the statement.

[Listing 18-2a](#) shows the Pascal program RepeatTest.

[Listing 18-2a:](#) Pascal program RepeatTest

```
001 PROGRAM RepeatTest;
002
003 VAR
004     number : integer;
005
006 FUNCTION sqroot(n : integer) : real;
007
008     CONST
009         epsilon = 1.0e-6;
010
011     VAR
012         r : real;
```

```
013
014     BEGIN
015         r := n;
016
017         REPEAT
018             r := (n/r + r)/2
019         UNTIL r*r - n < epsilon;
020
021         sqroot := r
022     END;
023
024 BEGIN {Calculate a square root using Newton&apos;s method.}
025     number := 1024;
026
027     writeln(&apos;The square root of &apos;, number:4);
028     writeln(&apos;    by standard sqrt() function: &apos;, sqrt(number):9:6);
029     writeln(&apos;  by declared sqroot() function: &apos;, sqroot(number):9:6);
030 END.

                30 source lines.
                 0 syntax errors.
              0.08 seconds total parsing time.

               112 instructions generated.
              0.06 seconds total code generation time.
```

[Listing 18-2b](#) shows the generated Jasmin object code for the REPEAT statement in function sqroot().

[Listing 18-2b:](#) The generated Jasmin object code for the REPEAT statement

```
.var 0 is n I
.var 1 is r F
.var 2 is sqroot F

...

.line 17
L001:
.line 18
    iload_0
    i2f
    fload_1
    fdiv
    fload_1
    fadd
    iconst_2
    i2f
    fdiv
    fstore_1
    fload_1
    fload_1
    fmul
    iload_0
    i2f
    fsub
    ldc     1.0E-6
    fcmpg
    iflt    L003
    iconst_0
    goto    L004
L003:
    iconst_1
L004:
    ifne    L002
    goto    L001
L002:
```

[Listing 18-3a](#) shows the Pascal program WhileTest, and [Listing 18-3b](#) shows the generated Jasmin object code for the WHILE statement in function sqroot().

[Listing 18-3a:](#) Pascal program WhileTest

```
001 PROGRAM WhileTest;
002
003 VAR
004     number : integer;
005
006 FUNCTION sqroot(n : integer) : real;
007
008     CONST
009         epsilon = 1.0e-6;
010
011     VAR
012         r : real;
013
014     BEGIN
015         r := n;
016
017         WHILE r*r - n > epsilon DO BEGIN
018             r := (n/r + r)/2
019         END;
020
021         sqroot := r
022     END;
023
024 BEGIN {Calculate a square root using Newton&apos;s method.}
025     number := 1024;
026
027     writeln(&apos;The square root of &apos;, number:4);
028     writeln(&apos;    by standard sqrt() function: &apos;, sqrt(number):9:6);
029     writeln(&apos;  by declared sqroot() function: &apos;, sqroot(number):9:6);
030 END.

                30 source lines.
                 0 syntax errors.
              0.09 seconds total parsing time.

               114 instructions generated.
              0.05 seconds total code generation time.
```

[Listing 18-3b:](#) The generated Jasmin object code for the WHILE statement

```
    .var 0 is n I
    .var 1 is r F
    .var 2 is sqroot F

    ...

    .line 17
L001:
    fload_1
    fload_1
    fmul
    iload_0
    i2f
    fsub
    ldc       1.0E-6
    fcmpg
    ifgt      L003
    iconst_0
    goto      L004
L003:
    iconst_1
L004:
    iconst_1
    ixor
    ifne      L002
    .line 18
    iload_0
    i2f
    fload_1
    fdiv
    fload_1
    fadd
    iconst_2
    i2f
    fdiv
    fstore_1
    goto      L001
L002:
```

Since a WHILE statement breaks out of the loop when the boolean test expression evaluates to false (value 0) the ICONST_1 and IXOR instructions after label L004 in Listing 18-3b perform the NOT of the boolean expression value at run time. The IFNE instruction, emitted as per the code template, then tests for the boolean value is not equal to 0 (i.e., the value is true). If so, the instruction branches out of the loop.

Listing 18-4a shows Pascal program ForTest, which includes some example FOR statements, and Listing 18-4b shows the generated Jasmin object code for the FOR statements.

Listing 18-4a: Pascal program ForTest

```
001 PROGRAM ForTest;
002
003 TYPE
004     grades = (A, B, C, D, F);
005
006 VAR
007     i, j, k, n : integer;
008     grade : grades;
009     ch : char;
010
011 BEGIN {FOR statements}
012     j  := 2;
013     ch := &apos;x&apos;;
014
015     FOR k := j TO 5 DO BEGIN
016         n := k;
017     END;
018
019     FOR k := n+1 DOWNTO j+2 DO i := k;
020
021     FOR i := 1 TO 2 DO BEGIN
022         FOR j := 1 TO 3 DO BEGIN
023             k := i*j;
024         END
025     END;
026
027     FOR grade := F DOWNTO a DO i := ord(grade);
028
029     FOR i := ord(ch) TO ord(&apos;z&apos;) DO j := i;
030 END.

                30 source lines.
                 0 syntax errors.
              0.09 seconds total parsing time.

               161 instructions generated.
              0.06 seconds total code generation time.
```

Listing 18-4b: The generated Jasmin object code for the FOR statements

```
.line 15
    getstatic         fortest/j I
    putstatic         fortest/k I
.line 15
L001:
    getstatic         fortest/k I
    iconst_5
    if_icmpgt         L003
    iconst_0
    goto      L004
L003:
    iconst_1
L004:
    ifne      L002
.line 16
    getstatic         fortest/k I
    putstatic         fortest/n I
```

```
        .line 15
        getstatic           fortest/k I
        iconst_1
        iadd
        putstatic           fortest/k I
        goto      L001
L002:
.line 19
        getstatic           fortest/n I
        iconst_1
        iadd
        putstatic           fortest/k I
.line 19
L005:
        getstatic           fortest/k I
        getstatic           fortest/j I
        iconst_2
        iadd
        if_icmplt           L007
        iconst_0
        goto      L008
L007:
        iconst_1
L008:
        ifne      L006
.line 19
        getstatic           fortest/k I
        putstatic           fortest/i I
.line 19
        getstatic           fortest/k I
        iconst_1
        isub
        putstatic           fortest/k I
        goto      L005
L006:
.line 21
        iconst_1
        putstatic           fortest/i I
.line 21
L009:
        getstatic           fortest/i I
        iconst_2
        if_icmpgt           L011
        iconst_0
        goto      L012
L011:
        iconst_1
L012:
        ifne      L010
.line 22
        iconst_1
        putstatic           fortest/j I
.line 22
L013:
        getstatic           fortest/j I
        iconst_3
        if_icmpgt           L015
        iconst_0
        goto      L016
L015:
        iconst_1
L016:
        ifne      L014
.line 23
        getstatic           fortest/i I
        getstatic           fortest/j I
        imul
        putstatic           fortest/k I
.line 22
        getstatic           fortest/j I
        iconst_1
        iadd
        putstatic           fortest/j I
        goto      L013
L014:
.line 21
        getstatic           fortest/i I
        iconst_1
        iadd
        putstatic           fortest/i I
        goto      L009
L010:
.line 27
        iconst_4
        putstatic           fortest/grade I
.line 27
L017:
        getstatic           fortest/grade I
        iconst_0
        if_icmplt           L019
        iconst_0
        goto      L020
L019:
        iconst_1
L020:
        ifne      L018
.line 27
        getstatic           fortest/grade I
        putstatic           fortest/i I
.line 27
        getstatic           fortest/grade I
        iconst_1
        isub
        putstatic           fortest/grade I
        goto      L017
L018:
.line 29
        getstatic           fortest/ch C
        putstatic           fortest/i I
.line 29
L021:
```

```
        getstatic        fortest/i I
        bipush  122
        if_icmpgt        L023
        iconst_0
        goto    L024
L023:
        iconst_1
L024:
        ifne    L022
.line 29
        getstatic        fortest/i I
        putstatic        fortest/j I
.line 29
        getstatic        fortest/i I
        iconst_1
        iadd
        putstatic        fortest/i I
        goto    L021
L022:
```
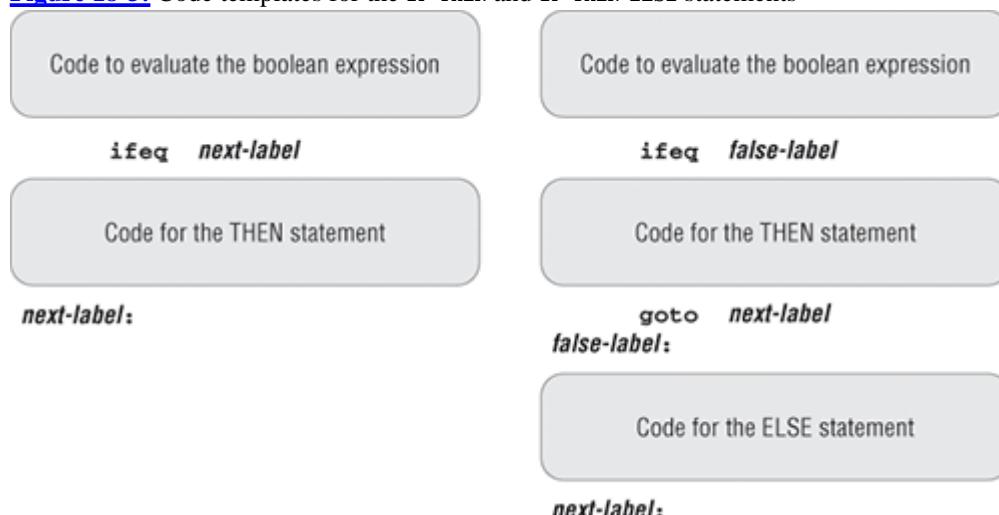
The Jasmin object code for a `FOR` statement may seem complex to generate, but all of the semantics of the statement is embodied in its parse tree, as built by class `ForStatementParser` in the front end. Class `LoopGenerator` in the compiler back end walks the tree and generates code for the statements, expressions, and tests.

The generated object code re-evaluates a `FOR` statement's initial and final control values each time through the loop. If you want the object code to evaluate those values only once each time the `FOR` statement executes, then you would need to modify class `LoopGenerator` to allocate temporary local variables to hold the values.

The `IF` Statement

[Figure 18-3](#) shows the code templates for `IF-THEN` and `IF-THEN-ELSE` statements.

Figure 18-3: Code templates for the `IF-THEN` and `IF-THEN-ELSE` statements

[Listing 18-5](#) shows the `generate()` method of the statement code generator class `IfGenerator`.

Listing 18-5: Method `generate()` of class `IfGenerator`

```
/**
 * Generate code for an IF statement.
 * @param node the root node of the statement.
 */
public void generate(ICodeNode node)
    throws PascalCompilerException
{
    ArrayList<ICodeNode> children = node.getChildren();
    ICodeNode expressionNode = children.get(0);
    ICodeNode thenNode = children.get(1);
    ICodeNode elseNode = children.size() > 2 ? children.get(2) : null;
    ExpressionGenerator expressionGenerator = new ExpressionGenerator(this);
    StatementGenerator  statementGenerator  = new StatementGenerator(this);
    Label nextLabel = Label.newLabel();

    // Generate code for the boolean expression.
    expressionGenerator.generate(expressionNode);

    // Generate code for a THEN statement only.
    if (elseNode == null) {
        emit(IFEQ, nextLabel);
        localStack.decrease(1);

        statementGenerator.generate(thenNode);
    }

    // Generate code for a THEN statement and an ELSE statement.
    else {
        Label falseLabel = Label.newLabel();

        emit(IFEQ, falseLabel);
        localStack.decrease(1);

        statementGenerator.generate(thenNode);
        emit(GOTO, nextLabel);

        emitLabel(falseLabel);
        statementGenerator.generate(elseNode);
    }

    emitLabel(nextLabel);
}
```

The `generate()` method generates Jasmin object code for either an IF-THEN or IF-THEN-ELSE statement according to the code template.

Listing 18-6a shows Pascal program `IfTest`, which contains example IF-THEN and IF-THEN-ELSE statements, and Listing 18-6b shows the generated Jasmin assembly object code for the IF statements.

Listing 18-6a: Pascal program `IfTest`

```
001 PROGRAM IfTest;
002
003 VAR
004     i, j, t, f : integer;
005
006 BEGIN {IF statements}
007     i := 3;  j := 4; t := 0; f := 0;
008
009     IF i < j THEN t := 300;
010
011     IF i = j THEN t := 200
012             ELSE f := -200;
013
014     {Cascading IF THEN ELSEs.}
015     IF     i = 1 THEN f := 10
016     ELSE IF i = 2 THEN f := 20
017     ELSE IF i = 3 THEN t := 30
018     ELSE IF i = 4 THEN f := 40
019     ELSE              f := -1;
020
021     {The "dandling ELSE".}
022     IF i = 3 THEN IF j = 2 THEN t := 500 ELSE f := -500;
023 END.

             23 source lines.
              0 syntax errors.
           0.17 seconds total parsing time.

            145 instructions generated.
           0.08 seconds total code generation time.
```

Listing 18-6b: The generated Jasmin object code for the IF statements

```
.line 9
     getstatic          iftest/i I
     getstatic          iftest/j I
     if_icmplt          L002
     iconst_0
     goto    L003
L002:
     iconst_1
L003:
     ifeq    L001
.line 9
     sipush  300
     putstatic          iftest/t I
L001:
.line 11
     getstatic          iftest/i I
     getstatic          iftest/j I
     if_icmpeq          L005
     iconst_0
     goto    L006
L005:
     iconst_1
L006:
     ifeq    L007
.line 11
     sipush  200
     putstatic          iftest/t I
     goto    L004
L007:
.line 12
     sipush  200
     ineg
     putstatic          iftest/f I
L004:
.line 15
     getstatic          iftest/i I
     iconst_1
     if_icmpeq          L009
     iconst_0
     goto    L010
L009:
     iconst_1
L010:
     ifeq    L011
.line 15
     bipush  10
     putstatic          iftest/f I
     goto    L008
L011:
.line 16
     getstatic          iftest/i I
     iconst_2
     if_icmpeq          L013
     iconst_0
     goto    L014
L013:
     iconst_1
L014:
     ifeq    L015
.line 16
     bipush  20
     putstatic          iftest/f I
     goto    L012
L015:
.line 17
     getstatic          iftest/i I
```

```
        iconst_3
        if_icmpeq        L017
        iconst_0
        goto      L018
 L017:
        iconst_1
 L018:
        ifeq      L019
.line 17
        bipush   30
        putstatic        iftest/t I
        goto      L016
 L019:
.line 18
        getstatic        iftest/i I
        iconst_4
        if_icmpeq        L021
        iconst_0
        goto      L022
 L021:
        iconst_1
 L022:
        ifeq      L023
.line 18
        bipush   40
        putstatic        iftest/f I
        goto      L020
 L023:
.line 19
        iconst_1
        ineg
        putstatic        iftest/f I
 L020:
 L016:
 L012:
 L008:
.line 22
        getstatic        iftest/i I
        iconst_3
        if_icmpeq        L025
        iconst_0
        goto      L026
 L025:
        iconst_1
 L026:
        ifeq      L024
.line 22
        getstatic        iftest/j I
        iconst_2
        if_icmpeq        L028
        iconst_0
        goto      L029
 L028:
        iconst_1
 L029:
        ifeq      L030
.line 22
        sipush   500
        putstatic        iftest/t I
        goto      L027
 L030:
.line 22
        sipush   500
        ineg
        putstatic        iftest/f I
 L027:
 L024:
```
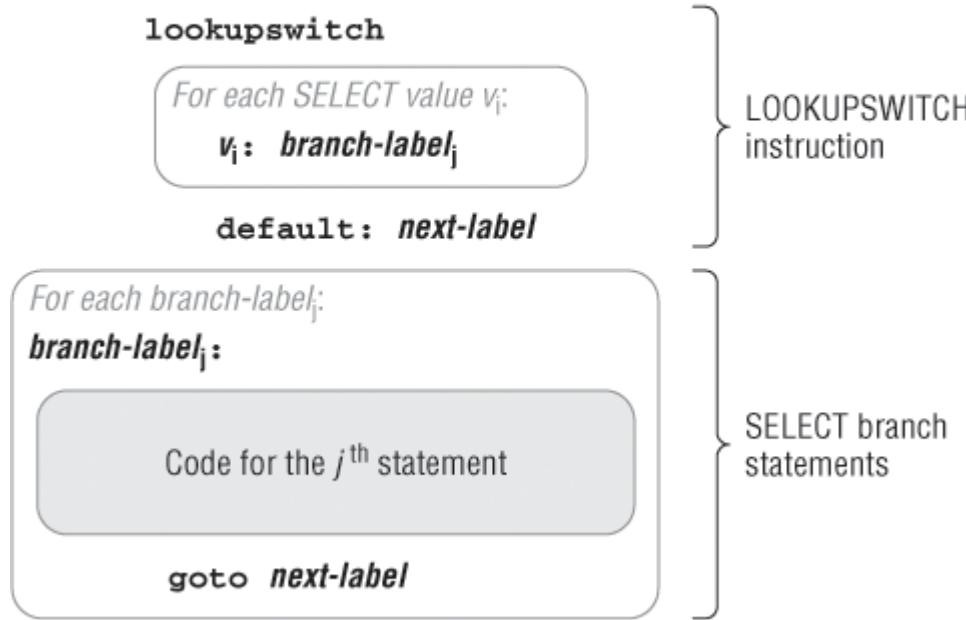
In the statements starting in lines 15 and 22, each nested IF is a separate statement that has its own "next" labels. See these labels L020, L016, L012, and L008 in the object code before line 22 and labels L027 and L024 at the end.

The SELECT Statement

As you first saw in Chapter 15, the Pascal compiler generates the LOOKUPSWITCH instruction for the Pascal CASE statement. Figure 18-4 shows the code template for the CASE statement, which a SELECT node represents in the parse tree.

Figure 18-4: Code template for the Pascal CASE statement

The template shows that code generation for a CASE statement is in three parts: code to evaluate the SELECT expression, the LOOKUPSWITCH instruction with its value-label pairs, and code for the SELECT branch statements. The compiler emits one value-label pair for each SELECT value ending with a default-label pair. The value-label pairs are sorted by their values, and more than one pair can have the same branch label. For each branch label, the compiler generates code for the corresponding SELECT branch statement preceded by the label. A GOTO instruction at the end of the code for each SELECT branch statement takes the execution out of the CASE statement.

Listing 18-7 shows the nested local class ValueLabelPair and the method generate() of the statement code generator subclass SelectGenerator. An array list of ValueLabelPair objects keeps track of the value-label pairs for the LOOKUPSWITCH instruction.

Listing 18-7: Nested local class `ValueLabelPair` and method `generate()` of class `SelectGenerator`

```
/**
 * A value-label pair for the LOOKUPSWITCH: A SELECT branch value
 *                                 and its statement label.
 */
private class ValueLabelPair
{
    private int value;
    private Label label;

    /**
     * Constructor.
     * @param value the SELECT branch value.
     * @param label the corresponding statement label.
     */
    private ValueLabelPair(int value, Label label)
    {
        this.value = value;
        this.label = label;
    }
}

/**
 * Generate code for a SELECT statement.
 * @param node the root node of the statement.
 */
public void generate(ICodeNode node)
    throws PascalCompilerException
{
    ArrayList<ICodeNode> selectChildren = node.getChildren();
    ICodeNode exprNode = selectChildren.get(0);
    ArrayList<Label> branchLabels = new ArrayList<Label>();

    branchLabels.add(Label.newLabel());  // "next" label

    // Generate code to evaluate the SELECT expression.
    ExpressionGenerator exprGenerator = new ExpressionGenerator(this);
    exprGenerator.generate(exprNode);

    // Process the select branches.
    ArrayList<ValueLabelPair> pairs = processSelectBranches(selectChildren,
                                                     branchLabels);

    // Generate code for the LOOKUPSWITCH and the branch statements.
    generateLookupSwitch(pairs, branchLabels);
    generateBranchStatements(selectChildren, branchLabels);
}
```

Method generate() creates the list of branch labels branchLabels and initializes it with the "next" label. It calls exprGenerator.generate() to generate code for the SELECT expression and leave its value on top of the operand stack. It calls processSelectBranches() to create the array list of ValueLabelPair objects, generateLookupSwitch() to generate the LOOKUPSWITCH instruction and its value-label pairs, and finally generateBranchStatements() to generate code for the SELECT branch statements.

[Listing 18-8:](#) Methods `processSelectBranches()` and `sortPairs()` of class `SelectGenerator`

```
/**
 * Process the SELECT branches.
 * @param selectChildren the child nodes of the SELECT node.
 * @param branchLabels the branch labels.
 * @return the array list of table entries.
 */
private ArrayList<ValueLabelPair> processSelectBranches
                                    (ArrayList<ICodeNode> selectChildren,
                                     ArrayList<Label> branchLabels)
{
    ArrayList<ValueLabelPair> pairs = new ArrayList<ValueLabelPair>();

    // Loop over the SELECT branches.
    for (int i = 1; i < selectChildren.size(); ++i) {
        ICodeNode branchNode = selectChildren.get(i);
        ICodeNode constantsNode = branchNode.getChildren().get(0);
        Label branchLabel = Label.newLabel();

        branchLabels.add(branchLabel);

        // Loop over the constants children of the branch&apos;s
        // SELECT_CONSTANTS node and create the label-value pairs.
        ArrayList<ICodeNode> constantsList = constantsNode.getChildren();
        for (ICodeNode constantNode : constantsList) {
            Object value = constantNode.getAttribute(VALUE);
            int intValue = constantNode.getType() == STRING_CONSTANT
                            ? (int) (((String) value).charAt(0))
                            : ((Integer) value).intValue();

            pairs.add(new ValueLabelPair(intValue, branchLabel));
        }
    }

    // Sort and return the list of value-label pairs.
    sortPairs(pairs);
    return pairs;
}

/**
 * Sort the label-value pairs into ascending order by value.
 * @param pairs the list of pairs.
 */
private void sortPairs(ArrayList<ValueLabelPair> pairs)
{
    // Simple insertion sort on the SELECT branch values.
    for (int i = 0; i < pairs.size()-1; ++i) {
        for (int j = i+1; j < pairs.size(); ++j) {
            if (pairs.get(i).value > pairs.get(j).value) {
                ValueLabelPair temp = pairs.get(i);
                pairs.set(i, pairs.get(j));
                pairs.set(j, temp);
            }
        }
    }
}
```

Method `processSelectBranches()` loops over the SELECT_BRANCH nodes and for each node, it creates a new branch statement label `branchLabel`, which it adds to the `branchLabels` list, and then it loops over the constant nodes of the SELECT branch. For each constant, it creates a new `ValueLabelPair` using the constant value and the `branchLabel`. Before returning the array list of `ValueLabelPair` objects, the method calls `sortPairs()` to sort the pairs by their values.

[Listing 18-9:](#) Methods `generateLookupSwitch()` and `generateBranchStatements()` of class `SelectGenerator`

```
/**
 * Generate code for the LOOKUPSWITCH instruction.
 * @param entries the table entries.
 * @param branchLabels the branch labels.
 */
private void generateLookupSwitch(ArrayList<ValueLabelPair> entries,
                                   ArrayList<Label> branchLabels)
{
    emitBlankLine();
    emit(LOOKUPSWITCH);

    // For each entry, emit the value and its label.
    for (ValueLabelPair entry : entries) {
        emitLabel(entry.value, entry.label);
    }

    // Emit the default label;
    emitLabel("default", branchLabels.get(0));

    localStack.decrease(1);
}

/**
 * Generate code for the branch statements.
 * @param selectChildren the child nodes of the SELECT node.
 * @param branchLabels the branch labels
 * @throws PascalCompilerException if an error occurred.
 */
private void generateBranchStatements(ArrayList<ICodeNode> selectChildren,
                                       ArrayList<Label> branchLabels)
    throws PascalCompilerException
{
    StatementGenerator stmtGenerator = new StatementGenerator(this);
    emitBlankLine();

    // Loop to emit each branch label and then generate code for
    // the corresponding branch statement.
    for (int i = 1; i < selectChildren.size(); ++i) {
        ICodeNode branchNode = selectChildren.get(i);
        ICodeNode statementNode = branchNode.getChildren().get(1);

        emitLabel(branchLabels.get(i));
        stmtGenerator.generate(statementNode);

        // Branch to the "next" label.
        emit(GOTO, branchLabels.get(0));
    }
```

```
          // Emit the "next" label.
          emitLabel(branchLabels.get(0));

          emitBlankLine();
      }
```

Method `generateLookupSwitch()` emits the LOOKUPSWITCH instruction and then loops over the list of `ValueLabelPair` objects to emit each value-label pair.

Method `generateBranchStatements()` loops over the SELECT_BRANCH nodes and uses the `branchLabels` list to emit each branch label and generate code for the corresponding SELECT branch statement. The method calls `stmtGenerator.generate()` and then emits a GOTO instruction to branch out of the CASE statement. After generating code for all the SELECT branch statements, it emits the "next" label.

Listing 18-10a shows Pascal program CaseTest containing a variety of CASE statements, including a nested CASE statement. Listing 18-10b shows the generated Jasmin object code for the CASE statements.

Listing 18-10a: Pascal program `CaseTest`

```
001 PROGRAM CaseTest;
002
003 TYPE
004     sizes = (small, medium, large);
005
006 VAR
007     i, j, even, odd, prime : integer;
008     ch, str : char;
009     size : sizes;
010
011 BEGIN {CASE statements}
012     i := 3;  ch := 'b';
013     size := medium;
014     even := -990; odd := -999; prime := 0;
015
016     CASE i+1 OF
017         1:      j := i;
018         -8:     j := 8*i;
019         5, 7, 4: j := 574*i;
020     END;
021
022     CASE ch OF
023         'c', 'b' : str := 'p';
024         'a'      : str := 'q'
025     END;
026
027     CASE size OF
028         small:  str := 's';
029         medium: str := 'm';
030         large:  str := 'l';
031     END;
032
033     FOR i := -5 TO 15 DO BEGIN
034         CASE i OF
035             2: prime := i;
036             -4, -2, 0, 4, 6, 8, 10, 12: even := i;
037             -3, -1, 1, 3, 5, 7, 9, 11:  CASE i OF
038                                             -3, -1, 1, 9:   odd := i;
039                                             2, 3, 5, 7, 11: prime := i;
040                                         END
041         END;
042     END
043 END.

                  43 source lines.
                   0 syntax errors.
                0.08 seconds total parsing time.

                 142 instructions generated.
                0.05 seconds total code generation time.
```

Listing 18-10b: The generated Jasmin object code for the CASE statements

```
.line 16
      getstatic       casetest/i I
      iconst_1
      iadd

      lookupswitch
        -8: L003
        1: L002
        4: L004
        5: L004
        7: L004
        default: L001

L002:
.line 17
      getstatic       casetest/i I
      putstatic       casetest/j I
      goto    L001
L003:
.line 18
      bipush   8
      getstatic       casetest/i I
      imul
      putstatic       casetest/j I
      goto    L001
L004:
.line 19
      sipush   574
      getstatic       casetest/i I
      imul
      putstatic       casetest/j I
      goto    L001
L001:

.line 22
      getstatic       casetest/ch C

      lookupswitch
        97: L007
        98: L006
```

```
        99: L006
        default: L005


L006:
.line 23
      bipush   112
      putstatic          casetest/str C
      goto     L005
L007:
.line 24
      bipush   113
      putstatic          casetest/str C
      goto     L005
L005:

.line 27
      getstatic          casetest/size I

      lookupswitch
        0: L009
        1: L010
        2: L011
        default: L008


L009:
.line 28
      bipush   115
      putstatic          casetest/str C
      goto     L008
L010:
.line 29
      bipush   109
      putstatic          casetest/str C
      goto     L008
L011:
.line 30
      bipush   108
      putstatic          casetest/str C
      goto     L008
L008:

.line 33
      iconst_5
      ineg
      putstatic          casetest/i I
.line 33
L012:
      getstatic          casetest/i I
      bipush   15
      if_icmpgt          L014
      iconst_0
      goto     L015
L014:
      iconst_1
L015:
      ifne     L013
.line 34
      getstatic          casetest/i I

      lookupswitch
        -4: L018
        -3: L019
        -2: L018
        -1: L019
        0: L018
        1: L019
        2: L017
        3: L019
        4: L018
        5: L019
        6: L018
        7: L019
        8: L018
        9: L019
        10: L018
        11: L019
        12: L018
        default: L016


L017:
.line 35
      getstatic          casetest/i I
      putstatic          casetest/prime I
      goto     L016
L018:
.line 36
      getstatic          casetest/i I
      putstatic          casetest/even I
      goto     L016
L019:
.line 37
      getstatic          casetest/i I

      lookupswitch
        -3: L021
        -1: L021
        1: L021
        2: L022
        3: L022
        5: L022
        7: L022
        9: L021
        11: L022
        default: L020


L021:
.line 38
      getstatic          casetest/i I
      putstatic          casetest/odd I
      goto     L020
L022:
```

```
.line 39
    getstatic       casetest/i I
    putstatic       casetest/prime I
    goto    L020
L020:

    goto    L016
L016:
```

This concludes code generation for Pascal's control statements.

Compiling Arrays and Subscripted Variables

You did code generation for Pascal strings in the previous chapter. Now complete the Pascal compiler with code generation for the other structured data types, arrays, and records.

Code generation for arrays has four parts:

- Code to allocate memory for an array variable.
- Code to allocate memory for each structured array element.
- Code for a subscripted variable in an expression.
- Code for a subscripted variable that is the target of an assignment statement.

Allocating Memory for Arrays

For a one-dimensional array, the generated object code first pushes the number of array elements onto the operand stack, and then the NEWARRAY (for scalar element values) or ANEWARRAY instruction (for structured element values) allocates memory for the array. For a multidimensional array, the generated object code must push the number of array elements of each dimension in order onto the operand stack. Then the MULTIANEWARRAY instruction allocates memory for the array.

If the array element values are structured, the generated code also allocates memory for each array element. Finally, the generated code stores the address of the array into the array variable.

Chapter 15 described the explicit operands of the NEWARRAY, ANEWARRAY, and MULTIANEWARRAY instructions to create arrays.

To accommodate generating object code to allocate memory for an array variable, you need to redo major portions of class StructuredDataGenerator.

Listing 18-11 shows methods generate() and generateAllocateData().

Listing 18-11: Methods `generate()` and `generateAllocateData()` of class `StructuredDataGenerator`

```
/**
 * Generate code to allocate the string, array, and record variables
 * of a program, procedure, or function.
 * @param routineId the routine&apos;s symbol table entry.
 */
public void generate(SymTabEntry routineId)
{
    SymTab symTab = (SymTab) routineId.getAttribute(ROUTINE_SYMTAB);
    ArrayList<SymTabEntry> ids = symTab.sortedEntries();

    // Loop over all the symbol table&apos;s identifiers to generate
    // data allocation code for string, array, and record variables.
    emitBlankLine();
    for (SymTabEntry id : ids) {
        if (id.getDefinition() == VARIABLE) {
            TypeSpec idType = id.getTypeSpec();

            if (isStructured(idType)) {
                TypeForm idForm = idType.getForm();

                if (idType.isPascalString() || (idForm == RECORD)) {
                    generateAllocateData(id, idType);
                }
                else {
                    generateAllocateArray(id, idType);
                }
            }
        }
    }
}

/**
 * Generate data to allocate a string, array, or record variable.
 * @param variableId the variable&apos;s symbol table entry.
 * @param type the variable&apos;s data type.
 */
private void generateAllocateData(SymTabEntry variableId, TypeSpec type)
{
    TypeForm form = type.getForm();

    if (type.isPascalString()) {
        generateAllocateString(variableId, type);
    }
    else if (form == ARRAY) {
        generateAllocateElements(variableId, type, 1);
    }
    else if (form == RECORD) {
        generateAllocateRecord(variableId, type);
    }
}
```

Method `generate()` loops over all the identifiers in a symbol table to look for variables of any structured data type — strings, arrays, or records. It calls `generateAllocateData()` for string and record variables, and `generateAllocateArray()` for array variables.

Depending on the structured data type, method `generateAllocateData()` calls either `generateAllocateString()`, `generateAllocateElements()`, or `generateAllocateRecord()`. You saw `generateAllocateString()` in the previous chapter, and you'll examine `generateAllocateRecord()` later in this chapter.

Listing 18-12 shows method `generateAllocateData()`.

```
/**
 * Generate code to allocate data for an array variable.
 * @param variableId the symbol table entry of the variable.
 * @param arrayType the array type.
 */
private void generateAllocateArray(SymTabEntry variableId,
                                   TypeSpec arrayType)
{
    TypeSpec elmtType = arrayType;
    int dimCount = 0;

    // Count the dimensions and emit a load constant of each size.
    emitBlankLine();
    do {
        int size = (Integer) elmtType.getAttribute(ARRAY_ELEMENT_COUNT);
        ++dimCount;
        emitLoadConstant(size);
        elmtType = (TypeSpec) elmtType.getAttribute(ARRAY_ELEMENT_TYPE);
    } while ((elmtType.getForm() == ARRAY) &&
               (! elmtType.isPascalString()));

    // The array element type.
    elmtType = elmtType.baseType();
    TypeForm elmtForm = elmtType.getForm();
    String typeName =
          elmtType == Predefined.integerType ? "int"
        : elmtType == Predefined.realType     ? "float"
        : elmtType == Predefined.booleanType ? "boolean"
        : elmtType == Predefined.charType     ? "char"
        : elmtForm == ENUMERATION             ? "int"
        : elmtForm == RECORD                  ? "java/util/HashMap"
        : elmtType.isPascalString()           ? "java/lang/StringBuilder"
        :                                       null;

    // One-dimensional array.
    if (dimCount == 1) {
        Instruction newArray = isStructured(elmtType) ? ANEWARRAY
                                                      : NEWARRAY;
        emit(newArray, typeName);
    }

    // Multidimensional array.
    else {
        emit(MULTIANEWARRAY, typeDescriptor(variableId.getTypeSpec()),
             Integer.toString(dimCount));
    }

    localStack.use(dimCount+1, dimCount);

    // Allocate data for structured elements.
    if (isStructured(elmtType)) {
        generateAllocateData(variableId, variableId.getTypeSpec());
    }

    // Store the allocation into the array variable.
    generateStoreData(variableId);
}
```

Method `generateAllocateArray()` generates instructions to load the size of each array dimension and then emits the `NEWARRAY`, `ANEWARRAY`, or `MULTIANEWARRAY` instruction. If the array elements are structured, then the method calls `generateAllocateData()` to allocate memory for each element. A call to `generateStoreData()` generates instructions to store the array address into the array variable.

Listing 18-13a shows Pascal program `AllocArrayTest1`, which contains one-, two-, and three-dimensional integer arrays.

Listing 18-13a: Pascal program `AllocArrayTest1`

```
PROGRAM AllocArrayTest1;

TYPE
    vector = ARRAY[0..9] OF integer;
    matrix = ARRAY[0..4, 0..3] OF integer;
    cube   = ARRAY[0..1, 0..2, 0..3] OF integer;

VAR
    a1 : vector;
    a2 : matrix;
    a3 : cube;

BEGIN
END.
```

Listing 18-13b shows the generated Jasmin object code to allocate memory for the array variables `a1`, `a2`, and `a3`. Because the array elements are scalar and not structured, no object code is necessary to allocate memory for the array elements.

Listing 18-13b: The generated Jasmin code to allocate memory for the arrays

```
.method public static main([Ljava/lang/String;)V

    ...

    bipush  10
    newarray int
    putstatic       allocarraytest1/a1 [I

    iconst_5
    iconst_4
    multianewarray  [[I 2
    putstatic       /a2 [[I

    iconst_2
    iconst_3
    iconst_4
    multianewarray  [[[I 3
    putstatic       allocarraytest1/a3 [[[I

    nop
```

```
        ...

    return

.limit locals 1
.limit stack 4
.end method
```

If the array element type is structured, then the generated object code must also allocate memory for each array element. Listing 18 -14a shows Pascal program `AllocArrayTest2`, which has string arrays.

Listing 18-14a: Pascal program `AllocArrayTest2`

```
PROGRAM AllocArrayTest2;

TYPE
    string = ARRAY[1..5] OF char;
    vector = ARRAY[0..9] OF string;
    matrix = ARRAY[0..4, 0..3] OF string;
    cube   = ARRAY[0..1, 0..2, 0..3] OF string;

VAR
    a1 : vector;
    a2 : matrix;
    a3 : cube;

BEGIN
END.
```

Since a string is a structured type, the generated object code must initialize each array element.

For the one-dimensional array variable `a1`, the generated Jasmin object code to allocate its elements is similar to the object code that the Java compiler would generate for the Java statement

```
for (int i = 0; i < 10; ++i) {
    a1[i] = PaddedString.create(5);
}
```

The generated Jasmin code to allocate the elements of the two-dimensional array `a2` is similar to

```
for (int i = 0; i < 5; ++i) {
    for (int j = 0; j < 4; ++j) {
        a2[i][j] = PaddedString.create(5);
    }
}
```

For the three-dimensional array `a3`, the generated Jasmin code to allocate its elements is similar to

```
for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 3; ++j) {
        for (int k = 0; k < 4; ++k) {
            a3[i][j][k] = PaddedString.create(5);
        }
    }
}
```

And so forth for higher dimensioned arrays.

In each of the above Java statements, the loop control variables `i`, `j`, and `k` are temporary.

Figure 18-5 shows the code template for allocating and initializing a one-dimensional array.

Figure 18-5: Code template for allocating and initializing a one-dimensional array

Instruction to load the array size.

NEWARRAY or **ANEWARRAY** instruction

```
        iconst_0
        istore_temp_index
loop_label:
        iload_temp_index
```


Instruction to load the array size.

```
        if_icmpge exit_label

        dup
        iload_temp_index
```


Code to load the element value

```
        Xastore
        iinc temp_index 1
        goto loop_label
exit_label:
```


Code to store the array address

The code generator allocates a temporary variable whose local variables array index is shown as *temp_index* in the code template. The temporary variable serves as the loop control variable. The DUP instruction ensures that each time through the loop, the address of the array is on the operand stack. At the bottom of the loop, the IINC instruction increments the temporary loop control variable by 1. After the final time through the loop, the array address is stored into the array variable.

Figure 18-6 shows the code template for allocating and initializing a multidimensional array.

For each dimension 1 through *n*-1 of an *n*-dimensional array, the generated code has a nested loop that executes the AALOAD instruction to load the value of each element, which is the address of the subarray in the next dimension. (If the array is a two-dimensional matrix, this would be the address of a row.) For the nth dimension, the code loops to initialize each element value as in the one-dimensional case.

Listing 18-14b shows the generated Jasmin object code that allocates memory for the array variables a1, a2, and a3 and for their elements.

Listing 18-14b: The generated Jasmin code to allocate memory for the arrays and their elements
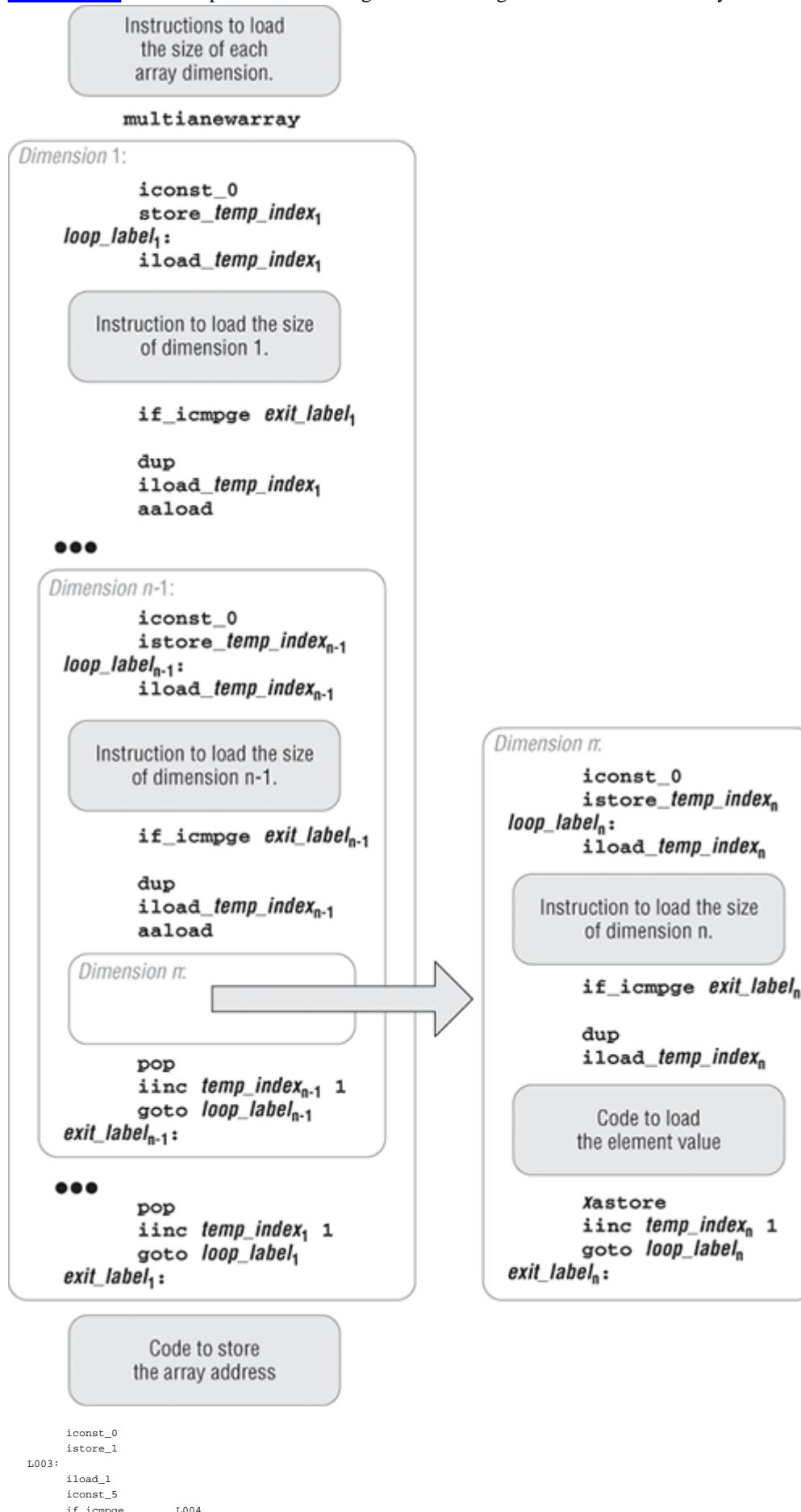
```
.method public static main([Ljava/lang/String;)V

    ...

    bipush   10
    anewarray        java/lang/StringBuilder

    iconst_0
    istore_1
L001:
    iload_1
    bipush   10
    if_icmpge        L002

    dup
    iload_1
    iconst_5
    invokestatic     PaddedString.create(I)Ljava/lang/StringBuilder;
```

```
        aastore
        iinc    1 1
        goto    L001
L002:
        putstatic          allocarraytest2/a1 [Ljava/lang/StringBuilder;

        iconst_5
        iconst_4
        multianewarray     [[Ljava/lang/StringBuilder; 2
```

Figure 18-6: Code template for allocating and initializing a multidimensional array


```
        iconst_0
        istore_1
L003:
        iload_1
        iconst_5
        if_icmpge          L004
```

```
            dup
            iload_1
            aaload

            iconst_0
            istore_2
L005:
            iload_2
            iconst_4
            if_icmpge      L006

            dup
            iload_2
            iconst_5
            invokestatic      PaddedString.create(I)Ljava/lang/StringBuilder;
            aastore
            iinc    2 1
            goto    L005
L006:
            pop
            iinc    1 1
            goto    L003
L004:
            putstatic         allocarraytest2/a2 [[Ljava/lang/StringBuilder;

            iconst_2
            iconst_3
            iconst_4
            multianewarray    [[[Ljava/lang/StringBuilder; 3

            iconst_0
            istore_1
L007:
            iload_1
            iconst_2
            if_icmpge         L008

            dup
            iload_1
            aaload

          iconst_0
            istore_2
L009:
            iload_2
            iconst_3
            if_icmpge         L010

            dup
            iload_2
            aaload

            iconst_0
            istore_3
L011:
            iload_3
            iconst_4
            if_icmpge         L012

            dup
            iload_3
            iconst_5
            invokestatic      PaddedString.create(I)Ljava/lang/StringBuilder;
            aastore
            iinc    3 1
            goto    L011
L012:
            pop
            iinc    2 1
            goto    L009
L010:
            pop
            iinc    1 1
            goto    L007
L008:
            putstatic         allocarraytest2/a3 [[[Ljava/lang/StringBuilder;

            nop

            ...

            return

.limit locals 4
.limit stack 7
.end method
```

The generated Jasmin code reserves three temporary local variables stored in slots 1, 2, and 3 of the local variables array. These temporary variables are re-used. For instance, the temporary variable in slot 1 is used to allocate the elements of a1, a2, and a3.

shows method generateAllocateElements() of class StructuredDataGenerator, which the compiler calls recursively to generate the object code to loop over each dimension of the array.

Listing 18-15: Method `generateAllocateElements()` of class `StructuredDataGenerator`

```java
    /**
     * Generate code that loops over the array dimensions to allocate
     * data for each element.
     * @param variableId the symbol table entry of the variable.
     * @param elmtType the data type of the array element.
     * @param dimensionIndex the first dimension is 1, second is 2, etc.
     */
    private void generateAllocateElements(SymTabEntry variableId,
                                          TypeSpec elmtType, int dimensionIndex)
    {
        int count = (Integer) elmtType.getAttribute(ARRAY_ELEMENT_COUNT);
        int tempIndex = localVariables.reserve();  // temporary loop variable
        Label loopStartLabel = Label.newLabel();
        Label loopExitLabel  = Label.newLabel();
```

```
    // Initialize temporary variable to 0.
    emitBlankLine();
    emitLoadConstant(0);
    emitStoreLocal(Predefined.integerType, tempIndex);

    // Top of loop: Compare the temporary variable to the element count.
    emitLabel(loopStartLabel);
    emitLoadLocal(Predefined.integerType, tempIndex);
    emitLoadConstant(count);
    emit(IF_ICMPGE, loopExitLabel);

    emitBlankLine();
    emit(DUP);  // duplicate dimension address for use by each element

    localStack.use(4, 3);

    elmtType = (TypeSpec) elmtType.getAttribute(ARRAY_ELEMENT_TYPE);

    // Allocate data for a string or record element.
    if (elmtType.isPascalString() || (elmtType.getForm() == RECORD)) {
        emitLoadLocal(Predefined.integerType, tempIndex);
        localStack.increase(1);

        generateAllocateData(null, elmtType);
    }

    // Allocate data for the next array dimension.
    else {
        emitLoadLocal(Predefined.integerType, tempIndex);
        emit(AALOAD);
        localStack.use(2);

        generateAllocateElements(variableId, elmtType, dimensionIndex+1);
    }

    // Bottom of loop: Increment the temporary variable and
    //                 branch back to the top of the loop.
    emit(IINC, tempIndex, 1);
    emit(GOTO, loopStartLabel);
    emitLabel(loopExitLabel);

    // Pop off the extraneous duplicated dimension address.
    if (dimensionIndex > 1) {
        emit(POP);
        localStack.decrease(1);
    }

    localVariables.release(tempIndex);
}
```

Method `generateAllocateElements()` first reserves a temporary loop variable and emits code to initialize to variable to 0. It then emits the code at the top of the loop, which compares the value of the loop variable against the number of array elements in the dimension.

For the heart of the loop, the method emits a DUP instruction to duplicate the address of the array dimension on the operand stack. This always leaves a copy of the address on the stack for each iteration to use. The method then either emits code to prepare to loop over the next dimension, or if it's the last dimension, the method calls `generateAllocateData()` to emit code that allocates memory for a string or record array element. If there is another dimension, the method emits the AALOAD instruction to load the address of the next dimension and calls itself recursively to process that dimension.

At the bottom of the loop, the method emits the IINC instruction to increment the value of the temporary loop variable and the GOTO instruction to branch back to the top of the dimension's loop.

Past the bottom of the loop, the method emits a POP instruction to remove the address of the array dimension from the operand stack, and then it releases the temporary loop variable for possible re-use.

Subscripted Variables in Expressions and Assignments

Subscripted variables are compiled by methods of class `ExpressionGenerator`, which you first saw in Chapter 16.

For a one-dimensional array variable in the expression of an assignment statement, such as

```
j := a1[i];
```

the compiler generates the Jasmin object code

```
getstatic        arraytest/a1 [I
getstatic        arraytest/i I
iaload
putstatic        arraytest/j I
```

The IALOAD instruction requires two implicit operand values on the operand stack: the address of the array a1 and subscript value i.

An example of an assignment statement with a multidimensional array variable in the expression is

```
n := a3[i, j, k];
```

which generates the Jasmin object code

```
getstatic        arraytest/a3 [[[I
getstatic        arraytest/i I
aaload
getstatic        arraytest/j I
aaload
getstatic        arraytest/k I
iaload
putstatic        arraytest/n I
```

The generated code uses the AALOAD instruction to load the subarray address of each dimension. The IALOAD instruction loads the element value from the last dimension's subarray.

Listing 18-16 shows a new version of method `generateLoadValue()`, which generates object code to load a variable's value onto the operand stack.

Listing 18-16: Method `generateLoadValue()` of class `ExpressionGenerator`

```
/**
```

```
 * Generate code to load a variable&apos;s value.
 * @param variableNode the variable node.
 */
protected void generateLoadValue(ICodeNode variableNode)
{
    ArrayList<ICodeNode> variableChildren = variableNode.getChildren();
    int childrenCount = variableChildren.size();

    // First load the variable&apos;s address (structured) or value (scalar).
    TypeSpec variableType = generateLoadVariable(variableNode);

    // Were there any subscript or field modifiers?
    if (childrenCount > 0) {
        ICodeNodeType lastModifierType =
                        variableChildren.get(childrenCount-1).getType();

        // Array subscript.
        if (lastModifierType == SUBSCRIPTS) {
            emitLoadArrayElement(variableType);
            localStack.decrease(1);
        }

        // Record field.
        else {
            emit(INVOKEVIRTUAL,
                "java/util/HashMap.get(Ljava/lang/Object;)" +
                "Ljava/lang/Object;");
            emitCheckCastClass(variableType);

            if (!isStructured(variableType)) {
                emit(INVOKEVIRTUAL, valueSignature(variableType));
            }

            localStack.decrease(1);
        }
    }
}
```

Method `generateLoadValue()` calls `generateLoadVariable()` to generate object code that loads either a variable's scalar value or the address of the variable's structured value onto the operand stack. If the variable has any modifiers (i.e., subscripts or fields), the method checks the last modifier. If it's a subscript, the method calls `emitLoadArrayElement()` of class `CodeGenerator` (see Listing 15-16), which emits the proper *x*ALOAD instruction to load the value of an array element.

Later in this chapter, you'll examine the case where the last modifier is a field.

shows method `generateLoadVariable()`.

** Method `generateLoadVariable()` of class `ExpressionGenerator`**

```
/**
 * Generate code to load a variable&apos;s address (structured) or
 * value (scalar).
 * @param variableNode the variable node.
 */
protected TypeSpec generateLoadVariable(ICodeNode variableNode)
{
    SymTabEntry variableId = (SymTabEntry) variableNode.getAttribute(ID);
    TypeSpec variableType = variableId.getTypeSpec();
    ArrayList<ICodeNode> variableChildren = variableNode.getChildren();
    int childrenCount = variableChildren.size();

    emitLoadVariable(variableId);
    localStack.increase(1);

    // Process subscripts and/or fields.
    for (int i = 0; i < childrenCount; ++i) {
        ICodeNode modifier = variableChildren.get(i);
        ICodeNodeType modifierType = modifier.getType();
        boolean last = i == childrenCount-1;

        if (modifierType == SUBSCRIPTS) {
            variableType = generateArrayElement(modifier, variableType,
                                                last);
        }
        else if (modifierType == ICodeNodeTypeImpl.FIELD) {
            variableType = generateRecordField(modifier, last);
        }
    }

    return variableType;
}
```

Method `generateLoadVariable()` calls `emitLoadVariable()` to emit object code to load a variable's value. If the variable has no modifiers, then the method's work is done. Otherwise, the method loops over the variable's subscript and field modifiers and calls either `generateArrayElement()` or `generateRecordField()`.

shows method `generateArrayElement()`, which generates code to load an array element.

** Method `generateArrayElement()` of class `ExpressionGenerator`**

```
/**
 * Generate code for a subscripted variable.
 * @param subscriptsNode the SUBSCRIPTS node.
 * @param variableType the array variable type.
 * @param last true if this is the variable&apos;s last subscript, else false.
 * @return the type of the element.
 */
private TypeSpec generateArrayElement(ICodeNode subscriptsNode,
                                      TypeSpec variableType, boolean last)
{
    ArrayList<ICodeNode> subscripts = subscriptsNode.getChildren();
    ICodeNode lastSubscript = subscripts.get(subscripts.size()-1);
    TypeSpec elmtType = variableType;

    for (ICodeNode subscript : subscripts) {
        generate(subscript);

        TypeSpec indexType =
                    (TypeSpec) elmtType.getAttribute(ARRAY_INDEX_TYPE);

        if (indexType.getForm() == SUBRANGE) {
            int min = (Integer) indexType.getAttribute(SUBRANGE_MIN_VALUE);
            if (min != 0) {
                emitLoadConstant(min);
                emit(ISUB);
```

```
                    localStack.use(1);
                }
            }

            if (!last || (subscript != lastSubscript)) {
                emit(AALOAD);
                localStack.decrease(1);
            }

            elmtType = (TypeSpec) elmtType.getAttribute(ARRAY_ELEMENT_TYPE);
        }

        return elmtType;
    }
```

Method `generateArrayElement()` loops over the subscripts. For each subscript, it calls `generate()` recursively to generate object code that evaluates the subscript expression and leaves its value on top of the operand stack. Since Jasmin array elements are always indexed starting with 0, the method generates code to subtract the minimum value of the array dimension's index type if necessary. Except for the last subscript, the method emits an AALOAD instruction to load the address of the next dimension's subarray.

After the last subscript, the generated code leaves the subarray address and the subscript value on top of the operand stack. The XALOAD instruction that method `generateLoadValue()` emits (see Listing 18-16) loads the scalar value of the array element.

When a one-dimensional array variable is the target of an assignment statement, such as

```
        al[i] := j;
```

the generated Jasmin object code is

```
        getstatic          arraytest/al [I
        getstatic          arraytest/i I
        getstatic          arraytest/j I
        iastore
```

The IASTORE instruction requires three implicit operand values on the operand stack: the address of the array al, the subscript value i, and the value j to be stored into the array.

An example of a multidimensional array variable as the target of an assignment statement is

```
        a3[i, j, k] := n;
```

which generates the Jasmin object code

```
        getstatic          arraytest/a3 [[[I
        getstatic          arraytest/i I
        aaload
        getstatic          arraytest/j I
        aaload
        getstatic          arraytest/k I
        getstatic          arraytest/n I
        iastore
```

Once again the AALOAD instruction loads the subarray address of each dimension. The final IASTORE instruction stores the value into an element of the last dimension's subarray.

Make changes to class `AssignmentGenerator` to generate object code that assigns a value to a subscripted variable. Listing 18-19 shows the changes to the last half of the `generate()` method.

Listing 18-19: Changes to method `generate()` of class `AssignmentGenerator`

```
    public void generate(ICodeNode node)
    {
        ...

        ArrayList<ICodeNode> targetChildren = targetNode.getChildren();
        int childrenCount = targetChildren.size();

        // The last modifier, if any, is the variable&apos;s last subscript or field.
        ICodeNode lastModifier = null;

        // The target variable has subscripts and/or fields.
        if (childrenCount > 0) {
            lastModifier = targetChildren.get(childrenCount-1);

            if (assignmentType.isPascalString()) {
                exprGenerator.generateLoadValue(targetNode);
            }
            else {
                assignmentType = exprGenerator.generateLoadVariable(targetNode);
            }
        }

        // Assign to a Pascal string array element or field.
        else if (assignmentType.isPascalString()) {
            emitLoadVariable(targetId);
            localStack.increase(1);
        }

        // Generate code to do the assignment.
        if (targetType.isPascalString()) {
            generateStringAssignment(assignmentType, exprNode,
                                    exprType, exprGenerator);
        }
        else {
            generateScalarAssignment(targetType, targetId, lastModifier,
                                    slot, nestingLevel, exprNode, exprType,
                                    exprGenerator);
        }
    }
```

If the target variable has subscript or field modifiers, method `generate()` sets `lastModifier` to the tree node of the variable's last modifier. It calls `exprGenerator.generateLoadValue()` for a string assignment or `exprGenerator.generateLoadVariable()` for any other type. Either call generates object code that leaves the address of the target variable on top of the operand stack. The call to `generateScalarAssignment()` now passes `lastModifier`.

Listing 18-20 shows changes to the last half of method `generateScalarAssignment()`.

Listing 18-20: Changes to method `generateScalarAssignment()` of class `AssignmentGenerator`

```
        private void generateScalarAssignment(TypeSpec targetType,
                                    SymTabEntry targetId,
                                    ICodeNode lastModifier,
                                    int index, int nestingLevel,
                                    ICodeNode exprNode,
                                    TypeSpec exprType,
                                    ExpressionGenerator exprGenerator)
    {
        ...

        // Generate code to store the expression value into the target variable.
        // The target variable has no subscripts or fields.
        if (lastModifier == null) {
            emitStoreVariable(targetId, nestingLevel, index);
            localStack.decrease(isWrapped(targetId) ? 2 : 1);
        }

        // The target variable is a field.
        else if (lastModifier.getType() == ICodeNodeTypeImpl.FIELD) {
            TypeSpec dataType = lastModifier.getTypeSpec().baseType();
            TypeForm typeForm = dataType.getForm();

            if ((typeForm == SCALAR) || (typeForm == ENUMERATION)) {
                emit(INVOKESTATIC, valueOfSignature(dataType));
            }

            emit(INVOKEVIRTUAL,
                "java/util/HashMap.put(Ljava/lang/Object;" +
                "Ljava/lang/Object;)Ljava/lang/Object;");
            emit(POP);
            localStack.decrease(3);
        }

        // The target variable is an array element.
        else {
            emitStoreArrayElement(targetType);
            localStack.decrease(3);
        }
    }
```

If the target variable is an array element, then method `generateScalarAssignment()` calls `emitStoreArrayElement()` in class `CodeGenerator` (see Listing 15-19), which emits the proper *x*ASTORE instruction to store the value of an array element. You'll examine later how `generateScalarAssignment()` generates code for an assignment to a record field.

Listing 18-21a shows Pascal program `ArrayTest`, which contains subscripted variables in expressions and as assignment targets.

Listing 18-21a: Pascal program `ArrayTest`

```
001 PROGRAM ArrayTest;
002
003 TYPE
004     vector = ARRAY[0..9] OF integer;
005     matrix = ARRAY[0..4, 0..4] OF integer;
006     cube   = ARRAY[0..1, 0..2, 0..3] OF integer;
007
008 VAR
009     i, j, k, n : integer;
010     a1         : vector;
011     a2         : matrix;
012     a3         : cube;
013
014 BEGIN
015     i := 0; j := 0; k := 0;
016
017     j := a1[i];
018     k := a2[i, j];
019     n := a3[i, j, k];
020
021     a1[i] := j;
022     a2[i, j] := k;
023     a3[i, j, k] := n;
024
025     a3[i][a1[j]][k] := a2[i][j] - a3[k, 2*n][k+1];
026 END.

              26 source lines.
               0 syntax errors.
            0.08 seconds total parsing time.

             123 instructions generated.
            0.05 seconds total code generation time.
```

Listing 18-21b shows the generated Jasmin object code for the assignment statements that contain subscripted variables.

Listing 18-21b: The generated Jasmin code for the statements that contain subscripted variables

```
.method public static main([Ljava/lang/String;)V

    ...

    bipush   10
    newarray  int
    putstatic        arraytest/a1 [I

    iconst_5
    iconst_5
    multianewarray   [[I 2
    putstatic        arraytest/a2 [[I

    iconst_2
    iconst_3
    iconst_4
    multianewarray   [[[I 3
    putstatic        arraytest/a3 [[[I

    ...

.line 17
    getstatic        arraytest/a1 [I
    getstatic        arraytest/i I
    iaload
```

```
        putstatic           arraytest/j I
.line 18
        getstatic           arraytest/a2 [[I
        getstatic           arraytest/i I
        aaload
        getstatic           arraytest/j I
        iaload
        putstatic           arraytest/k I
.line 19
        getstatic           arraytest/a3 [[[I
        getstatic           arraytest/i I
        aaload
        getstatic           arraytest/j I
        aaload
        getstatic           arraytest/k I
        iaload
        putstatic           arraytest/n I
.line 21
        getstatic           arraytest/a1 [I
        getstatic           arraytest/i I
        getstatic           arraytest/j I
        iastore
.line 22
        getstatic           arraytest/a2 [[I
        getstatic           arraytest/i I
        aaload
        getstatic           arraytest/j I
        getstatic           arraytest/k I
        iastore
.line 23
        getstatic           arraytest/a3 [[[I
        getstatic           arraytest/i I
        aaload
        getstatic           arraytest/j I
        aaload
        getstatic           arraytest/k I
        getstatic           arraytest/n I
        iastore
.line 25
        getstatic           arraytest/a3 [[[I
        getstatic           arraytest/i I
        aaload
        getstatic           arraytest/a1 [I
        getstatic           arraytest/j I
        iaload
        aaload
        getstatic           arraytest/k I
        getstatic           arraytest/a2 [[I
        getstatic           arraytest/i I
        aaload
        getstatic           arraytest/j I
        iaload
        getstatic           arraytest/a3 [[[I
        getstatic           arraytest/k I
        aaload
        iconst_2
        getstatic           arraytest/n I
        imul
        aaload
        getstatic           arraytest/k I
        iconst_1
        iadd
        iaload
        isub
        iastore

        ...

        return

    .limit locals 1
    .limit stack 6
    .end method
```

The generated Jasmin code for line 25 handles a subscript expression that includes a subscripted variable. The first implicit stack operand of final IASTORE instruction is the address of the array a3, which was pushed onto the operand stack by the initial GETSTATIC instruction.

Compiling Records and Record Fields

Implement the value of a Pascal record variable at run time with a `java.util.HashMap` object. The name of each record field will serve as the key to access the corresponding field value.[1]

---

[1] Perhaps a more obvious implementation of record values is with instances of local classes with fields that have the same names and types as the record types. Such a solution would require that you solve some naming issues, especially with nested or unnamed Pascal record types.

For example, consider the following Pascal type declarations and variable declarations:

```
PROGRAM RecordTest;

TYPE
    String2  = ARRAY [0..1 ] OF char;
    String5  = ARRAY [0..4 ] OF char;
    String8  = ARRAY [0..7 ] OF char;
    String16 = ARRAY [0..15] OF char;

    AddressRec = RECORD
                    street : String16;
                    city   : String16;
                    state  : String2;
                    zip    : String5;
```

```
                END;

    PersonRec =  RECORD
                    firstName : String16;
                    lastName  : String16;
                    age       : integer;
                    address   : AddressRec;
                    phones    : ARRAY [0..1] OF String8;
                 END;

VAR
    john : PersonRec;
```

The Pascal compiler implements each `PersonRec` and `AddressRec` value as a Java hash map. The following Java statements allocate the hash maps:

```
private static HashMap john;

...

    // Allocate john.
    john = new HashMap();
    john.put("firstName", PaddedString.create(16));
    john.put("lastName", PaddedString.create(16));
    john.put("age", 0);

    // Allocate john.address.
    john.put("address", new HashMap());
    ((HashMap) john.get("address")).put("street", PaddedString.create(16));
    ((HashMap) john.get("address")).put("city", PaddedString.create(16));
    ((HashMap) john.get("address")).put("state", PaddedString.create(2));
    ((HashMap) john.get("address")).put("zip", PaddedString.create(5));

    // Allocate john.phones.
    john.put("phones", new StringBuilder[2]);
    ((StringBuilder[]) john.get("phones"))[0] = PaddedString.create(8);
    ((StringBuilder[]) john.get("phones"))[1] = PaddedString.create(8);
```

The Pascal compiler compiles the Pascal assignment statements

```
john.firstName := 'John';
john.age := 24;
john.address.street := '1680 25th Street';
john.phones[0]       := '111-1111';
```

similarly to way the Java compiler compiles the Java assignment statements

```
((StringBuilder) john.get("firstName")).setLength(0);
john.put("age", 24);
((StringBuilder) ((HashMap) john.get("address")).get("street")).
    setLength(0);
((StringBuilder) ((HashMap) john.get("address")).get("street")).
    append("1680 25th Steet");
((StringBuilder[]) john.get("phones"))[0].setLength(0);
((StringBuilder[]) john.get("phones"))[0].append("111-1111");
```

Allocating Records

Listing 18-22 shows method `generateAllocateRecord()` of class `StructuredDataGenerator`.

Listing 18-22: Class `generateAllocateRecord()` of class `StructuredDataGenerator`

```
/**
 * Generate code to allocate a record variable as a HashMap.
 * @param variableId the symbol table entry of the variable.
 * @param recordType the record data type.
 */
private void generateAllocateRecord(SymTabEntry variableId,
                                    TypeSpec recordType)
{
    SymTab recordSymTab = (SymTab) recordType.getAttribute(RECORD_SYMTAB);

    // Allocate a hash map.
    emitBlankLine();
    emit(NEW, "java/util/HashMap");
    emit(DUP);
    emit(INVOKENONVIRTUAL, "java/util/HashMap/<init>()V");
    localStack.use(2, 1);

    // Allocate the record fields.
    generateAllocateFields(recordSymTab);

    // Store the allocation into the record variable.
    generateStoreData(variableId);
}
```

Method `generateAllocateRecord()` emits the object code to create a new `java.util.HashMap` object for a Pascal record value and the code to call the object's constructor. It then calls `generateAllocateFields()` to generate code to allocate the fields.

Listing 18-23 shows method `generateAllocateFields()`.

Listing 18-23: Method `generateAllocateFields()` of class `StructuredDataGenerator`

```
/**
 * Generate code to allocate array and record variables and fields.
 * @param symTab the record's symbol table containing fields.
 */
protected void generateAllocateFields(SymTab symTab)
{
    ArrayList<SymTabEntry> fields = symTab.sortedEntries();

    // Loop over all the symbol table's field identifiers
    // to generate data allocation code.
    for (SymTabEntry fieldId : fields) {
        TypeSpec idType = fieldId.getTypeSpec().baseType();

        // Load the field name.
        emit(DUP);
        emitLoadConstant(fieldId.getName());
        localStack.increase(2);
```

```
            // Allocate data for a string, array, or record field.
            if (isStructured(idType)) {
                TypeForm idForm = idType.getForm();

                if (idType.isPascalString() || (idForm == RECORD)) {
                    generateAllocateData(fieldId, idType);
                }
                else {
                    generateAllocateArray(fieldId, idType);
                }
            }

            // Initialize a scalar field.
            else {
                Object value = BackendFactory.defaultValue(idType);

                if (idType == Predefined.integerType) {
                    emitLoadConstant((Integer) value);
                }
                else if (idType == Predefined.realType) {
                    emitLoadConstant((Float) value);
                }
                else if (idType == Predefined.booleanType) {
                    emitLoadConstant((Boolean) value ? 1 : 0);
                }
                else /* idType == Predefined.charType */ {
                    emitLoadConstant((Character) value);
                }

                // Promote a scalar value to an object.
                emit(INVOKESTATIC, valueOfSignature(idType));
                localStack.increase(1);

                generateStoreData(fieldId);
            }
        }
    }
```

Method `generateAllocateFields()` loops over the fields in the record type's symbol table. For each field that is itself a record type or a string, the method calls `generateAllocateData()`, and for a field that's an array, it calls `generateAllocateArray()`. If the field is a scalar, the method calls the appropriate `emitLoadConstant()` method to generate a load instruction of the default value for the field type. Because only object values can be stored into a hash map, `generateAllocateFields()` emits an `INVOKESTATIC` instruction to call the `valueOf()` method of the corresponding object data type in order to promote the scalar value to an object, such as `Integer.valueOf()`. The method calls `generateStoreData()` to emit the code that stores the object value into the field.

Given the type definitions in the Pascal program `RecordTest` shown above, Listing 18-24 shows the object code the Pascal compiler generates to allocate and initialize the record value for program variable `john`.

Listing 18-24: The generated Jasmin code to allocate and initialize the record value for variable `john`

```
    new       java/util/HashMap
    dup
    invokenonvirtual  java/util/HashMap/<init>()V
    dup
    ldc       "address"

    new       java/util/HashMap
    dup
    invokenonvirtual  java/util/HashMap/<init>()V
    dup
    ldc       "city"
    bipush  16
    invokestatic      PaddedString.create(I)Ljava/lang/StringBuilder;
    invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                            ;)Ljava/lang/Object;
    pop
    dup
    ldc       "state"
    iconst_2
    invokestatic      PaddedString.create(I)Ljava/lang/StringBuilder;
    invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                            ;)Ljava/lang/Object;
    pop
    dup
    ldc       "street"
    bipush  16
    invokestatic      PaddedString.create(I)Ljava/lang/StringBuilder;
    invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                            ;)Ljava/lang/Object;
    pop
    dup
    ldc       "zip"
    iconst_5
    invokestatic      PaddedString.create(I)Ljava/lang/StringBuilder;
    invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                            ;)Ljava/lang/Object;
    pop
    invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                            ;)Ljava/lang/Object;
    pop
    dup
    ldc       "age"
    iconst_0
    invokestatic      java/lang/Integer.valueOf(I)Ljava/lang/Integer;
    invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                            ;)Ljava/lang/Object;
    pop
    dup

    ldc       "firstname"
    bipush  16
    invokestatic      PaddedString.create(I)Ljava/lang/StringBuilder;
    invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                            ;)Ljava/lang/Object;
    pop
    dup
    ldc       "lastname"
    bipush  16
    invokestatic      PaddedString.create(I)Ljava/lang/StringBuilder;
    invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                            ;)Ljava/lang/Object;
```

```
        pop
        dup
        ldc       "phones"

        iconst_2
        anewarray         java/lang/StringBuilder

        iconst_0
        istore_1
L003:
        iload_1
        iconst_2
        if_icmpge         L004

        dup
        iload_1
        bipush  8
        invokestatic      PaddedString.create(I)Ljava/lang/StringBuilder;
        aastore
        iinc    1 1
        goto    L003
L004:
        invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                                ;)Ljava/lang/Object;

        pop
        putstatic         recordtest/john Ljava/util/HashMap;
```

The generated Jasmin object code creates `java.util.HashMap` objects for the `PersonRec` value and the nested `AddressRec` value and initializes their fields. The code also allocates and initializes the `phones` string array field.

Record Fields in Expressions and Assignments

Setting and accessing record field values at run time involve calls to the `get()` and `put()` methods of the `java.util.HashMap` objects. As you saw above, before setting the value of a scalar field, the generated object code must first promote the value to an object by calling, for example, `Integer.valueOf()`. Conversely, when accessing the value of a scalar field, the generated code must demote the object value back to a scalar by calling, for example, `Integer.intValue()`.

Listing 18-25 shows method `generateRecordField()` of class `ExpressionGenerator`. Method `generateLoadVariable()` calls this method (see Listing 18-17).

Listing 18-25: Method `generateRecordField()` of class `ExpressionGenerator`

```
/**
 * Generate code to access the value of a record field.
 * @param fieldNode the FIELD node.
 * @param last true if this is the variable&apos;s last field, else false.
 * @return the type of the field.
 */
private TypeSpec generateRecordField(ICodeNode fieldNode, boolean last)
{
    SymTabEntry fieldId = (SymTabEntry) fieldNode.getAttribute(ID);
    String fieldName = fieldId.getName();
    TypeSpec fieldType = fieldNode.getTypeSpec();

    emitLoadConstant(fieldName);
    localStack.increase(1);

    if (!last) {
        emit(INVOKEVIRTUAL,
            "java/util/HashMap.get(Ljava/lang/Object;)" +
            "Ljava/lang/Object;");
        emitCheckCast(fieldType);
        localStack.decrease(1);
    }

    return fieldType;
}
```

After emitting the `INVOKEVIRTUAL` call to the hash map's `get()` method, method `generateRecordField()` emits a `CHECKCAST` instruction to verify that the object value obtained from the hash map is of the right type.

Method `generateRecordField()` is called to generate code for a record field in an expression and for a record field that is the target of an assignment. Therefore, it does nothing for the *last* field of a variable. For example, the method generates code for field `address` but not for field `street` of the variable `person.address.street`. The generated code needs to call the hash map `get()` method if the variable is in an expression or the hash map `put()` method if the variable is an assignment target. This decision is made elsewhere, as you'll see.

You saw method `generateLoadValue()` of class `ExpressionGenerator` in its entirety in Listing 18-16. When a record field is in and expression, the following statements of that method

```
            // Record field.
            else {
                emit(INVOKEVIRTUAL,
                    "java/util/HashMap.get(Ljava/lang/Object;)" +
                    "Ljava/lang/Object;");
                emitCheckCastClass(variableType);

                if (!isStructured(variableType)) {
                    emit(INVOKEVIRTUAL, valueSignature(variableType));
                }

                localStack.decrease(1);
            }
```

generate the code to call the hash map `get()` method to load the value of a field.

Continuing the example started above, if you add the following variable declarations:

```
age    : integer;
name   : String16;
street : String16;
```

then the assignment statements

```
age    := john.age;
name   := john.firstName;
street := john.address.firstName;
```

generate the Jasmin object code

```
. line 72
getstatic recordtest/john Ljava/util/HashMap;
ldc "age"
invokevirtual  java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/Object;
checkcast java/lang/Integer
invokevirtual java/lang/Integer.intValue()I
putstatic recordtest/age I
.line 73
getstatic recordtest/name Ljava/lang/StringBuilder;
dup
iconst_0
invokevirtual java/lang/StringBuilder.setLength(I)V
getstatic recordtest/john Ljava/util/HashMap;
ldc "firstname"
invokevirtual  java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/Object;
checkcast java/lang/StringBuilder
invokevirtual java/lang/StringBuilder.append(Ljava/lang/CharSequence;)L
                                        java/lang/StringBuilder;
pop
.line 74
getstatic recordtest/street Ljava/lang/StringBuilder;
dup
iconst_0
invokevirtual java/lang/StringBuilder.setLength(I)V
getstatic recordtest/john Ljava/util/HashMap;
ldc "address"
invokevirtual  java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/Object;
checkcast java/util/HashMap
ldc "street"
invokevirtual  java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/Object;
checkcast java/lang/StringBuilder
invokevirtual java/lang/StringBuilder.append(Ljava/lang/CharSequence;)L
                                        java/lang/StringBuilder;
pop
```

As shown earlier in <u>Listing 18-19</u>, the `generate()` method of class `AssignmentGenerator` calls `exprGenerator.generateLoadVariable()`. The following statements from that class's `generateScalarAssignment()` method, as first shown in <u>Listing 18-20</u>, generate the object code for a record field that is an assignment target:

```
// The target variable is a field.
else if (lastModifier.getType() == ICodeNodeTypeImpl.FIELD) {
    TypeSpec dataType = lastModifier.getTypeSpec().baseType();
    TypeForm typeForm = dataType.getForm();

    if ((typeForm == SCALAR) || (typeForm == ENUMERATION)) {
        emit(INVOKESTATIC, valueOfSignature(dataType));
    }

    emit(INVOKEVIRTUAL,
        "java/util/HashMap.put(Ljava/lang/Object;" +
        "Ljava/lang/Object;)Ljava/lang/Object;");
    emit(POP);
    localStack.decrease(3);
}
```

These statements emit the `INVOKESTATIC` call to the `valueOf()` method to promote a scalar value to an object and the `INVOKEVIRTUAL` call to the hash map `put()` method.

The Pascal assignment statements

```
john.lastName  := &apos;Doe&apos;;
john.age := 24;
john.address.street := &apos;1680 25th Street&apos;;
```

generate the Jasmin object code

```
.line 63
getstatic         recordtest/john Ljava/util/HashMap;
ldc      "lastname"
invokevirtual  java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/Object;
checkcast         java/lang/StringBuilder
dup
iconst_0
invokevirtual     java/lang/StringBuilder.setLength(I)V
ldc      "Doe"
invokevirtual     java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/l
                                            ang/StringBuilder;
bipush   16
iconst_3
invokestatic      PaddedString.blanks(II)Ljava/lang/StringBuilder;
invokevirtual     java/lang/StringBuilder.append(Ljava/lang/CharSequence;)L
                                            java/lang/StringBuilder;
pop
.line 64
getstatic         recordtest/john Ljava/util/HashMap;
ldc      "age"
bipush   24
invokestatic      java/lang/Integer.valueOf(I)Ljava/lang/Integer;
invokevirtual     java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/Object
                                            ;)Ljava/lang/Object;
pop
.line 65
getstatic         recordtest/john Ljava/util/HashMap;
ldc      "address"
invokevirtual  java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/Object;
checkcast         java/util/HashMap
ldc      "street"
invokevirtual     java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/Object;
checkcast         java/lang/StringBuilder
dup
iconst_0
invokevirtual     java/lang/StringBuilder.setLength(I)V
ldc      "1680 25th Street"
invokevirtual     java/lang/StringBuilder.append(Ljava/lang/String;)Ljava/l
                                            ang/StringBuilder;
pop
```

The generated code for line 64 includes the `INVOKEVIRTUAL` call to `java.util.HashMap.put()`. The generated code for lines 63 and 65 are string assignments and therefore have `INVOKEVIRTUAL` calls to methods `java.lang.StringBuilder.setLength()` and `java.lang.StringBuilder.append()` to set string values that were already "put" into the fields when the record value for `john` was first allocated and initialized (see <u>Listing 18-24</u>).

Suppose you have an array of record variables:

```
persons : ARRAY [0..4] OF PersonRec;
```

Then the assignment statement involving a composition of arrays and fields

```
persons[2].phones[0] := persons[3].phones[1];
```

generates the Jasmin code

```
getstatic          recordtest/persons [Ljava/util/HashMap;
iconst_2
aaload
ldc      "phones"
invokevirtual  java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/Object;
checkcast         [Ljava/lang/StringBuilder;
iconst_0
aaload
dup
iconst_0
invokevirtual     java/lang/StringBuilder.setLength(I)V
getstatic          recordtest/persons [Ljava/util/HashMap;
iconst_3
aaload
ldc      "phones"
invokevirtual  java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/Object;
checkcast         [Ljava/lang/StringBuilder;
iconst_1
aaload
invokevirtual java/lang/StringBuilder.append(Ljava/lang/CharSequence;)L
                                          java/lang/StringBuilder;
pop
```

This concludes the development of the Pascal compiler. You can now successfully compile and execute source programs written in the subset of the Pascal language you've been using in this book.

Program 18-1: Pascal Compiler III

Now do a major end-to-end test run of the Pascal compiler with a nontrivial source program. Program `Xref` reads a text file, prints a listing of the text, parses the text to find all the words, and prints a cross-reference listing. The cross-reference listing prints all the words in alphabetical order and for each word, the line numbers of the lines in the input text that contain that word.

Invoke the compiler with a command line similar to

```
java -classpath classes Pascal compile Xref.pas
```

This generates the output shown in
Listing 18-26a.

Listing 18-26a: Pascal program `xref`

```
001 PROGRAM Xref (input, output);
002
003    {Generate a cross-reference listing from a text file.}
004
005 CONST
006    MaxWordLength    =   20;
007    WordTableSize    =  500;
008    NumberTableSize  = 2000;
009    MaxLineNumber    =  999;
010
011 TYPE
012    charIndexRange   = 1..MaxWordLength;
013    wordIndexRange   = 1..WordTableSize;
014    numberIndexRange = 0..NumberTableSize;
015    lineNumberRange  = 1..MaxLineNumber;
016
017    wordType = ARRAY [charIndexRange] OF char;  {string type}
018
019    wordEntryType = RECORD  {entry in word table}
020                       word : wordType;      {word string}
021                       firstNumberIndex,   {head and tail of   }
022                       lastNumberIndex    {  linked number list}
023                            : numberIndexRange;
024                    END;
025
026    numberEntryType = RECORD  {entry in number table}
027                         number : lineNumberRange; {line number}
028                         nextIndex              {next index in}
029                              : numberIndexRange;  {  linked list}
030                      END;
031
032    wordTableType   = ARRAY [wordIndexRange]  OF wordEntryType;
033    numberTableType = ARRAY [numberIndexRange] OF numberEntryType;
034
035 VAR
036    wordTable                 : wordTableType;
037    numberTable               : numberTableType;
038    nextWordIndex             : wordIndexRange;
039    nextNumberIndex           : numberIndexRange;
040    lineNumber                : lineNumberRange;
041    wordTableFull, numberTableFull : boolean;
042    newLine                   : boolean;
043
044
045 FUNCTION NextChar : char;
046
047    {Fetch and echo the next character.
048     Print the line number before each new line.}
049
050    CONST
051       blank = ' ';
052
053    VAR
054       ch : char;
```

```
055
056     BEGIN
057         newLine := eoln;
058         IF newLine THEN BEGIN
059             readln;
060             writeln;
061             lineNumber := lineNumber + 1;
062             write(lineNumber:5, ' : ');
063         END;
064         IF newLine OR eof THEN BEGIN
065             ch := blank;
066         END
067         ELSE BEGIN
068             read(ch);
069             write(ch);
070         END;
071         NextChar := ch;
072     END;
073
074
075 FUNCTION IsLetter(ch : char) : boolean;
076
077     {Return true if the character is a letter, false otherwise.}
078
079     BEGIN
080         IsLetter :=    ((ch >= 'a') AND (ch <= 'z'))
081                     OR ((ch >= 'A') AND (ch <= 'Z'));
082     END;
083
084
085 FUNCTION ReadWord(VAR buffer : wordType) : boolean;
086
087     {Extract the next word and place it into the buffer.}
088
089     CONST
090         blank = ' ';
091
092     VAR
093         charcount : integer;
094         ch : char;
095
096     BEGIN
097         ReadWord := false;
098         ch := ' ';
099
100         {Skip over any preceding non-letters.}
101         IF NOT eof THEN BEGIN
102             REPEAT
103                 ch := NextChar;
104             UNTIL eof OR IsLetter(ch);
105         END;
106
107         {Find a letter?}
108         IF NOT eof THEN BEGIN
109             charcount := 0;
110
111             {Place the word's letters into the buffer.
112              Downshift uppercase letters.}
113             WHILE IsLetter(ch) DO BEGIN
114                 IF charcount < MaxWordLength THEN BEGIN
115                     IF (ch >= 'A') AND (ch <= 'Z') THEN BEGIN
116                         ch := chr(ord(ch) + (ord('a') - ord('A')));
117                     END;
118                     charcount := charcount + 1;
119                     buffer[charcount] := ch;
120                 END;
121                 ch := NextChar;
122             END;
123
124             {Pad the rest of the buffer with blanks.}
125             FOR charcount := charcount + 1 TO MaxWordLength DO BEGIN
126                 buffer[charcount] := blank;
127             END;
128
129             ReadWord := true;
130         END;
131     END;
132
133
134 PROCEDURE AppendLineNumber(VAR entry : wordEntryType);
135
136     {Append the current line number to the end of the
137      current word entry's linked list of numbers.}
138
139     BEGIN
140         IF nextNumberIndex < NumberTableSize THEN BEGIN
141
142             {entry.lastnumberindex is 0 if this is the word's
143              first number. Otherwise, it is the number table index
144              of the last number in the list, which we now extend
145              for the new number.}
146             IF entry.lastNumberIndex = 0 THEN BEGIN
147                 entry.firstNumberIndex := nextNumberIndex;
148             END
149             ELSE BEGIN
150                 numberTable[entry.lastNumberIndex].nextIndex :=
151                                             nextNumberIndex;
152             END;
153
154             {Set the line number at the end of the list.}
155             numberTable[nextNumberIndex].number    := lineNumber;
156             numberTable[nextNumberIndex].nextIndex := 0;
157             entry.lastNumberIndex := nextNumberIndex;
158             nextNumberIndex       := nextNumberIndex + 1;
159         END
160         ELSE BEGIN
161             numberTableFull := true;
162         END;
163     END;
164
165
```

```
166 PROCEDURE EnterWord;
167
168     {Enter the current word into the word table. Each word is first
169      read into the end of the table.}
170
171     VAR
172         i : wordIndexRange;
173
174     BEGIN
175         {By the time we process a word at the end of an input line,
176          lineNumber has already been incremented, so temporarily
177          decrement it.}
178         IF newLine THEN lineNumber := lineNumber - 1;
179
180         {Search to see if the word had already been entered
181          previously. Each time it&apos;s read in, it&apos;s placed at the end
182          of the word table.}
183         i := 1;
184         WHILE wordTable[i].word <> wordTable[nextWordIndex].word DO
185         BEGIN
186             i := i + 1;
187         END;
188
189         {Entered previously:  Update the existing entry.}
190         IF i < nextWordIndex THEN BEGIN
191             AppendLineNumber(wordTable[i]);
192         END
193
194         {New word:  Initialize the entry at the end of the table.}
195         ELSE IF nextWordIndex < WordTableSize THEN BEGIN
196             wordTable[i].lastNumberIndex := 0;
197             AppendLineNumber(wordTable[i]);
198             nextWordIndex := nextWordIndex + 1;
199         END
200
201         {Oops.  Table overflow!}
202         ELSE wordTableFull := true;
203
204         IF newLine THEN lineNumber := lineNumber + 1;
205     END;
206
207
208 PROCEDURE SortWords;
209
210     {Sort the words alphabetically.}
211
212     VAR
213         i, j : wordIndexRange;
214         temp : wordEntryType;
215
216     BEGIN
217         FOR i := 1 TO nextWordIndex - 2 DO BEGIN
218             FOR j := i + 1 TO nextWordIndex - 1 DO BEGIN
219                 IF wordTable[i].word > wordTable[j].word THEN BEGIN
220                     temp := wordTable[i];
221                     wordTable[i] := wordTable[j];
222                     wordTable[j] := temp;
223                 END;
224             END;
225         END;
226     END;
227
228
229 PROCEDURE PrintNumbers(i : numberIndexRange);
230
231     {Print a word&apos;s linked list of line numbers.}
232
233     BEGIN
234         REPEAT
235             write(numberTable[i].number:4);
236             i := numberTable[i].nextIndex;
237         UNTIL i = 0;
238         writeln;
239     END;
240
241
242 PROCEDURE PrintXref;
243
244     {Print the cross reference listing.}
245
246     VAR
247         i : wordIndexRange;
248
249     BEGIN
250         writeln;
251         writeln;
252         writeln(&apos;Cross-reference&apos;);
253         writeln(&apos;---------------&apos;);
254         writeln;
255         SortWords;
256         FOR i := 1 TO nextWordIndex - 1 DO BEGIN
257             write(wordTable[i].word);
258             PrintNumbers(wordTable[i].firstNumberIndex);
259         END;
260     END;
261
262
263 BEGIN {Xref}
264     wordTableFull   := false;
265     numberTableFull := false;
266     nextWordIndex   := 1;
267     nextNumberIndex := 1;
268     lineNumber      := 1;
269     write(&apos;   1 : &apos;);
270
271     {First read the words.}
272     WHILE NOT (eof OR wordTableFull OR numberTableFull) DO BEGIN
273
274         {Read each word into the last entry of the word table
275          and then enter it into its correct location.}
276         IF ReadWord(wordtable[nextwordIndex].Word) THEN BEGIN
```

```
277          EnterWord;
278        END;
279      END;
280
281    {Then print the cross reference listing if all went well.}
282    IF wordTableFull THEN BEGIN
283        writeln;
284        writeln(&apos;*** The word table is not large enough. ***&apos;);
285    END
286    ELSE IF numberTableFull THEN BEGIN
287        writeln;
288        writeln(&apos;*** The number table is not large enough. ***&apos;);
289    END
290    ELSE BEGIN
291        PrintXref;
292    END;
293
294    {Print final stats.}
295    writeln;
296    writeln((nextWordIndex - 1):5,   &apos; word entries.&apos;);
297    writeln((nextNumberIndex - 1):5, &apos; line number entries.&apos;);
298 END {Xref}.


               298 source lines.
                 0 syntax errors.
              0.20 seconds total parsing time.

               924 instructions generated.
              0.08 seconds total code generation time.
```

Listing 18-26b shows excerpts from the generated Jasmin object code.

Listing 18-26b: Excerpts from the generated Jasmin code for Pascal program `xref`

```
.class public xref
.super java/lang/Object

...

.method private static appendlinenumber(Ljava/util/HashMap;)V

.var 0 is entry Ljava/util/HashMap;

.line 140
     getstatic xref/nextnumberindex I
     sipush    2000
     if_icmplt L031
     iconst_0
     goto      L032
L031:
     iconst_1
L032:
     ifeq L033
.line 146
     aload_0
     ldc         "lastnumberindex"
     invokevirtual java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/
                                                          Object;
     checkcast    java/lang/Integer
     invokevirtual java/lang/Integer.intValue()I
     iconst_0
     if_icmpeq L035
     iconst_0
     goto L036
L035:
     iconst_1
L036:
     ifeq L037
.line 147
     aload_0
     ldc         "firstnumberindex"
     getstatic    xref/nextnumberindex I
     invokestatic java/lang/Integer.valueOf(I)Ljava/lang/Integer;
     invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                         Object;)Ljava/lang/Object;
     pop
     goto L034
L037:
.line 150
     getstatic    xref/numbertable [Ljava/util/HashMap;
     aload_0
     ldc         "lastnumberindex"
     invokevirtual java/util/HashMap.get(Ljava/lang/Object;)Ljava/lang/
                                                          Object;
     checkcast    java/lang/Integer
     invokevirtual java/lang/Integer.intValue()I
     aaload
     ldc         "nextindex"
     getstatic    xref/nextnumberindex I
     invokestatic java/lang/Integer.valueOf(I)Ljava/lang/Integer;
     invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                         Object;)Ljava/lang/Object;
     pop

L034:
.line 155
     getstatic    xref/numbertable [Ljava/util/HashMap;
     getstatic    xref/nextnumberindex I
     aaload
     ldc         "number"
     getstatic    xref/linenumber I
     invokestatic java/lang/Integer.valueOf(I)Ljava/lang/Integer;
     invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                         Object;)Ljava/lang/Object;
     pop
.line 156
     getstatic    xref/numbertable [Ljava/util/HashMap;
     getstatic    xref/nextnumberindex I
     aaload
     ldc         "nextindex"
```

```
        iconst_0
        invokestatic  java/lang/Integer.valueOf(I)Ljava/lang/Integer;
        invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                              Object;)Ljava/lang/Object;
        pop
.line 157
        aload_0
        ldc             "lastnumberindex"
        getstatic     xref/nextnumberindex I
        invokestatic  java/lang/Integer.valueOf(I)Ljava/lang/Integer;
        invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                              Object;)Ljava/lang/Object;
        pop
.line 158
        getstatic     xref/nextnumberindex I
        iconst_1
        iadd
        dup
        iconst_0
        sipush        2000
        invokestatic RangeChecker/check(III)V
        putstatic     xref/nextnumberindex I
        goto L030
L033:
.line 161
        iconst_1
        putstatic     xref/numbertablefull Z
L030:

        return

.limit locals 1
.limit stack 4
.end method

...

.method public static main([Ljava/lang/String;)V

        new RunTimer
        dup
        invokenonvirtual RunTimer/<init>()V
        putstatic        xref/_runTimer LRunTimer;
        new PascalTextIn
        dup
        invokenonvirtual PascalTextIn/<init>()V
        putstatic         xref/_standardIn LPascalTextIn;

        sipush    2001
        anewarray java/util/HashMap

        iconst_0
        istore_1
L071:
        iload_1
        sipush 2001
        if_icmpge L072

        dup
        iload_1

        new java/util/HashMap
        dup
        invokenonvirtual java/util/HashMap/<init>()V
        dup
        ldc             "nextindex"
        iconst_0
        invokestatic  java/lang/Integer.valueOf(I)Ljava/lang/Integer;
        invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                              Object;)Ljava/lang/Object;
        pop
        dup
        ldc             "number"
        iconst_0
        invokestatic  java/lang/Integer.valueOf(I)Ljava/lang/Integer;
        invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                              Object;)Ljava/lang/Object;
        pop
        aastore
        iinc 1 1
        goto L071
L072:
        putstatic xref/numbertable [Ljava/util/HashMap;

        sipush    500
        anewarray java/util/HashMap

        iconst_0
        istore_1
L073:
        iload_1
        sipush 500
        if_icmpge L074

        dup
        iload_1

        new java/util/HashMap
        dup
        invokenonvirtual java/util/HashMap/<init>()V
        dup
        ldc             "firstnumberindex"
        iconst_0
        invokestatic  java/lang/Integer.valueOf(I)Ljava/lang/Integer;
        invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                              Object;)Ljava/lang/Object;
        pop
        dup
        ldc             "lastnumberindex"
        iconst_0
        invokestatic  java/lang/Integer.valueOf(I)Ljava/lang/Integer;
```

```
        invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                                Object;)Ljava/lang/Object;
        pop
        dup
        ldc     "word"
        bipush 20
        invokestatic  PaddedString.create(I)Ljava/lang/StringBuilder;
        invokevirtual java/util/HashMap.put(Ljava/lang/Object;Ljava/lang/
                                                Object;)Ljava/lang/Object;
        pop
        aastore
        iinc 1 1
        goto L073
L074:
        putstatic xref/wordtable [Ljava/util/HashMap;

.line 264
        iconst_0
        putstatic xref/wordtablefull Z
.line 265
        iconst_0
        putstatic xref/numbertablefull Z
.line 266
        iconst_1
        dup
        iconst_1
        sipush 500
        invokestatic RangeChecker/check(III)V
        putstatic    xref/nextwordindex I
...

        return

.limit locals 2
.limit stack 8
.end method
```

A command line similar to

```
java -jar \jasmin-2.3\jasmin.jar xref.j
```

runs the Jasmin assembler to assemble the generated Jasmin code file `xref.j` and produce file `xref.class` in the current directory. Then a command line similar to

```
java -classpath .;PascalRTL.jar xref < Xref.pas
```

runs the compiled `xref` program with the source program itself as input. [Listing 18-27](#) shows some of the output.

[Listing 18-27:](#) Some of the output from running the compiled `xref` program

```
    1 : PROGRAM Xref (input, output);
    2 :
    3 :       {Generate a cross-reference listing from a text file.}
    4 :
    5 : CONST
    6 :       MaxWordLength    =   20;
    7 :       WordTableSize    =  500;
    8 :       NumberTableSize  = 2000;
    9 :       MaxLineNumber    =  999;
...
  298 : END {Xref}.
  299 :

Cross-reference
---------------

a                      3   3   77  80  81 107 115 116 116 175 231
all                  281
alphabetically       210
already              176 180
an                   175
and                   21  47  80  81  87 115 275
any                  100
append               136
appendlinenumber     134 191 197
array                 17  32  33
at                   154 175 181 194
been                 176 180
before                48
...
xref                   1 263 298
z                     80  81 115

  179 word entries.
  785 line number entries.

            1.31 seconds total execution time.
```

Chapter 19

Additional Topics

This chapter presents a brief overview of topics not covered in the previous chapters. Textbooks and advanced books on compiler writing discuss these topics in greater detail.

Scanning

The scanner that you developed in the early chapters used separate token classes. Their methods extracted and recognized the various Pascal tokens from the source program. While such scanners are easy to write and understand, they tend to require many lines of code and can run slowly.

A *table-driven scanner* can be more compact and run faster. To create such a scanner, you need to know a bit of the basics of *automata*.

Deterministic Finite Automata (DFA)

If you use the notation `<letter>` and `<digit>` to represent any letter character and any digit character, repectively, the *regular expression*

`<letter> ( <letter> | <digit> )*`
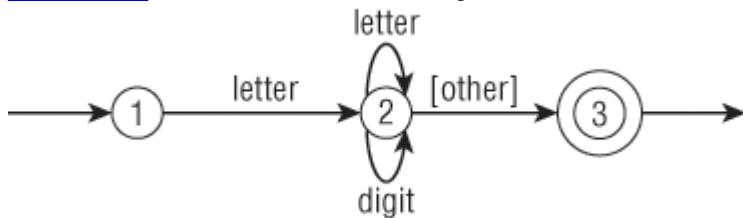
specifies that a Pascal identifier consists of a letter followed by zero or more letters or digits. The vertical bar separates alternatives and the star indicates zero or more occurrences of the preceding parenthesized subexpression. The angle brackets, parentheses, vertical bar, and star are *metasymbols* that have special meanings in regular expressions.

You can build a *finite automaton*, or *finite-state machine*, that implements this regular expression.

Figure 19-1 diagrams this finite automaton.

Figure 19-1: A finite automaton that recognizes a Pascal identifier

Each numbered circle in Figure 19-1 prepresents a *state* of the automaton. The state numbered 1 is the *start state* since it has the incoming arrow, and the state numbered 3, drawn with double lines, is an *accepting state*, where the recognition process ends. An automaton can have multiple accepting states. Each labeled arrow is a *transition* from one state to another that is taken for the type of input character in the label.

This automaton is a *deterministic finite automaton*, or DFA, because at each state, the next input character uniquely determines which transition to take to the next state.

The DFA in Figure 19-1 works to accept, or recognize, a Pascal identifier: Start in state 1. If the input character is a letter, transition to state 2. For each letter or digit character, transition back to state 2. When you encounter a character that is neither a letter nor a digit, transition to state 3 and accept the identifier.
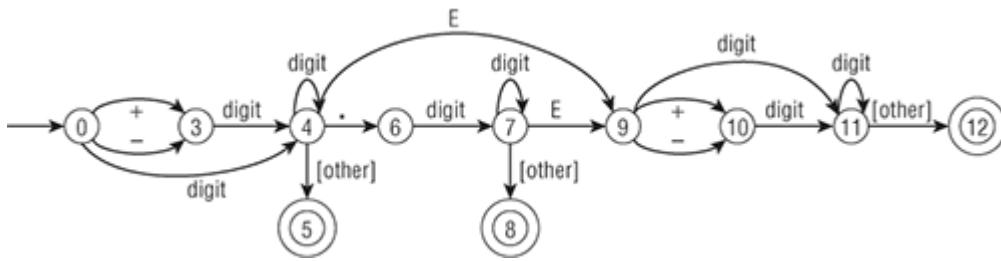
You can represent the behavior of this DFA by a *state-transition table*:

	Input character		
State	**Letter**	**Digit**	**other**
1	2		
2	2	2	3
3			

Each row of the table represents a state, and each column represents an input character type. The number in a cell indicates, for a given state, which state to transition to next for a given type of input character. Thus, in state 2, a letter or a digit keeps you in state 2, but any other character takes you to state 3. Cells without numbers take you to an error state.

Figure 19-2 shows a DFA that accepts a Pascal number, including any leading + or – sign.
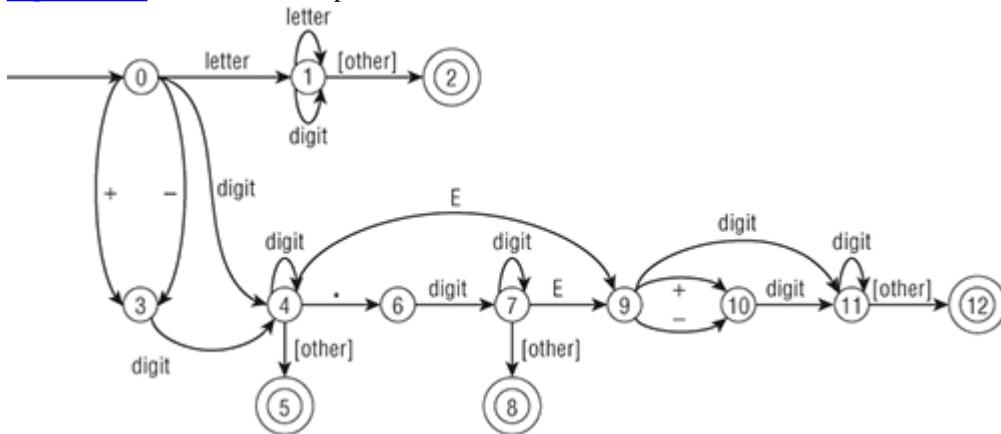
Figure 19-2: A DFA that accepts a Pascal number

This DFA has three accepting states. State 5 accepts an integer. State 8 accepts a real number without an exponent. State 12 accepts a real number with an exponent.

Figure 19-3 combines the two DFAs into one that accepts both Pascal identifiers and Pascal numbers.

Figure 19-3: A DFA that accepts both Pascal identifiers and numbers

Table-Driven Scanners

Listing 19-1 shows the key fields and methods of a simple table-driven scanner based on the DFA in Figure 19-3.

Listing 19-1: Key fields and methods of a simple table-driven scanner

```
/**
 * <h1>SimpleDFAScanner</h1>
 *
 * <p>A simple DFA scanner that recognizes Pascal identifers and numbers.</p>
 */
public class SimpleDFAScanner
{
    // Input characters.
    private static final int LETTER = 0;
    private static final int DIGIT  = 1;
    private static final int PLUS   = 2;
    private static final int MINUS  = 3;
    private static final int DOT    = 4;
    private static final int E      = 5;
    private static final int OTHER  = 6;

    // Error state.
    private static final int ERR = -99999;

    // State-transition matrix (acceptance states < 0)
    private static final int matrix[][] = {

        /*        letter digit   +     -     .     E other */
        /*  0 */ {   1,    4,    3,    3, ERR,    1, ERR },
        /*  1 */ {   1,    1,   -2,   -2,   -2,    1,  -2 },
        /*  2 */ { ERR,  ERR,  ERR, ERR, ERR, ERR, ERR },
        /*  3 */ { ERR,    4,  ERR, ERR, ERR, ERR, ERR },
        /*  4 */ {  -5,    4,   -5,  -5,    6,    9,  -5 },
        /*  5 */ { ERR,  ERR,  ERR, ERR, ERR, ERR, ERR },
        /*  6 */ { ERR,    7,  ERR, ERR, ERR, ERR, ERR },
        /*  7 */ {  -8,    7,   -8,  -8,   -8,    9,  -8 },
        /*  8 */ { ERR,  ERR,  ERR, ERR, ERR, ERR, ERR },
        /*  9 */ { ERR,   11,   10,  10, ERR, ERR, ERR },
        /* 10 */ { ERR,   11,  ERR, ERR, ERR, ERR, ERR },
        /* 11 */ { -12,   11,  -12, -12, -12, -12, -12 },
        /* 12 */ { ERR,  ERR,  ERR, ERR, ERR, ERR, ERR },
    };

    private char ch;    // current input character
    private int state;  // current state

    /**
     * Extract the next token from the source file.
     * @return name of the next token
     * @throws Exception if an error occurs.
     */
    private String nextToken()
        throws Exception
    {
        // Skip blanks.
        while (Character.isWhitespace(ch)) {
            nextChar();
        }

        // At EOF?
        if (ch == 0) {
            return null;
        }

        state = 0;  // start state
```

```
        StringBuilder buffer = new StringBuilder();

        // Loop to do state transitions.
        while (state >= 0) {   // not acceptance state
            state = matrix[state][typeOf(ch)];  // transition

            if ((state >= 0) || (state == ERR)) {
                buffer.append(ch);  // build token string
                nextChar();
            }
        }

        return buffer.toString();
    }

    /**
     * Scan the source file.
     * @throws Exception if an error occurs.
     */
    private void scan()
        throws Exception
    {
        nextChar();

        while (ch != 0) {  // EOF?
            String token = nextToken();

            if (token != null) {
                System.out.print("=====> \"" + token + "\" ");
                String tokenType =
                      (state ==  -2) ? "IDENTIFIER"
                    : (state ==  -5) ? "INTEGER"
                    : (state ==  -8) ? "REAL (fraction only)"
                    : (state == -12) ? "REAL"
                    :                  "**** ERROR ****";
                System.out.println(tokenType);
            }
        }
    }

    /**
     * Return the character type.
     * @param ch the character.
     * @return the type.
     */
    int typeOf(char ch)
    {
        return  (ch == 'E')            ? E
              : Character.isLetter(ch) ? LETTER
              : Character.isDigit(ch)  ? DIGIT
              : (ch == '+')            ? PLUS
              : (ch == '-')            ? MINUS
              : (ch == '.')            ? DOT
              :                          OTHER;
    }

    ...
}
```

Field `matrix`, the state-transition matrix, codes all the acceptance states as negative numbers. The heart of the scanner is the second `while` loop in method `nextToken()`. The statement

```
        state = matrix[state][typeOf(ch)];
```

makes the transition from one state to another.

As you can see, a table-driven scanner can be very fast. The key is the state-transition matrix. If the table is not too large and sparse,[1] the scanner code can also be small. There are software tools, including the compiler-compilers described later, that can automatically generate a state-transition matrix given the specifications (such as regular expressions) for the tokens.

---

[1] There are various techniques to store a sparse matrix compactly.

Syntax Notation

Throughout this book, syntax diagrams specified the syntax of Pascal statements. A more traditional syntax notation is the text-based Backus-Naur Form (BNF) named after its co-developers John Backus and Peter Naur.

Backus-Naur Form (BNF)

Like the regular expression you saw above, BNF has some metasymbols, shown in the the following table.

`::=`	"is defined as"
`|`	"or"
`< >`	Surround names of nonterminal (not literal) items

Use alternates in BNF to specify optional items. For example, an expression is a simple expression optionally followed by a relational operator followed by another simple expression:

```
<expression>  ::=  <simple expression>
                |  <simple expression> <rel op> <simple expression>
```

Use recursion to specify repeated items. For example, a digit sequence is a digit followed by zero or more digits:

```
<digit sequence>  ::=  <digit>
                    |  <digit> <digit sequence>
```

As another example, the BNF specification for the Pascal `IF` statement is

```
<if statement>  ::=  IF <expression> THEN <statement>
                  |  IF <expression> THEN <statement>
                             ELSE <statement>
```

Extended BNF (EBNF)

Extended BNF (EBNF) adds more metasymbols:

{ }	Surround items that are repeated zero or more times
[ ]	Surround optional items

Some examples in EBNF:

```
<digit sequence> ::= <digit> { <digit> }
<expression> ::=  <simple expression> [ <rel op> <simple expression> ]
<if statement> ::= IF <expression>  THEN <statement>
                                [ ELSE <statement> ]
```

Grammars and Languages

A *grammar* is the set of syntax rules, whether expressed in BNF, EBNF, or syntax diagrams. A *language* is the set of all token strings that are valid according to the grammar. Therefore, a grammar defines a language. A string of tokens that is valid is a syntactically correct statement.

Looking at this from the other direction, a statement is syntactically correct for the language defined by the grammar if the grammar can *derive* the statement. Each grammar rule *produces* valid token strings of the language and therefore, a grammar rule is also called a *production* or a *production rule*. The set of productions applied sequentially to a syntactically correct statement is the *derivation* of the statement.

Parsing

A parser can be *top down* or *bottom up*. This book showed how to develop a top-down parser.

Top-Down Parsers

In a top-down parser, you start with the topmost nonterminal grammar symbol such as `<program>` and work your way down recursively, hence the more complete name is *top-down recursive-descent parser*. Such a parser is easy to understand and write, but generally, they are big and slow.

As you know from the earlier chapters, you can write a parse method for each production rule. Each method expects to see source program tokens that match its rule. (You patterned the parse methods after the syntax diagrams.) A method may call other parse methods that implement lower production rules. For example, method `parse()` of class `AssignmentStatementParser` calls method `parse()` of class `ExpressionParser` which in turn calls method `parse()` of class `VariableParser`.

A parse is successful (no syntax errors) if it can derive its entire input string (the source program) from the production rules.

Bottom-Up Parsers

A bottom-up parser starts with tokens from the source program and works its way from the lower production rules up to the top. If it reaches the topmost grammar symbol, say `<program>`, the parse is successful. Thus, while a top-down parser starts with the topmost grammar symbol, a bottom-up parser tries to end with that symbol.

A popular type of bottom-up parser is the *shift-reduce parser*. It uses a set of production rules and a *parse stack,* which starts out empty.

The parser shifts (pushes) each input token it receives from the scanner onto the parse stack. When what's on top of the stack matches the right side of a production rule (what's to the right of the `::=`), the parser pops off the matching symbols and *reduces* (replaces) them with the nonterminal grammar symbol at the left side of the production rule. The parser tries to match the longest possible production rule.

The parser repeats the shift and reduce actions until the topmost grammar symbol is left on top of the parse stack. When that happens, the parse is successful and the parser accepts its input string.

Here's a simple example. Suppose you have the production rules

```
<expression> ::= <simple expression>
<simple expression> ::= <term> + <term>
<term> ::= <factor>  |  <factor> * <factor>
<factor> ::= <variable>
<variable> ::= <identifier>
<identifier> ::= a | b | c
```

The highest grammar symbol is `<expression>`. The following table shows how a shift-reduce parser parses the input string

```
a + b*c
```

Parse stack (top at right)	Input	Action
	a + b*c	shift
a	+ b*c	reduce
`<identifier>`	+ b*c	reduce
`<variable>`	+ b*c	reduce
`<factor>`	+ b*c	reduce
`<term>`	+ b*c	shift
`<term>` +	b*c	shift
`<term>` + b	*c	reduce

`<term> + <identifier>`	`*c`	reduce
`<term> + <variable>`	`*c`	reduce
`<term> + <factor>`	`*c`	shift
`<term> + <factor> *`	`c`	shift
`<term> + <factor> * c`		reduce
`<term> + <factor> * <identifier>`		reduce
`<term> + <factor> * <variable>`		reduce
`<term> + <factor> * <factor>`		reduce
`<term> + <term>`		reduce
`<simple expression>`		reduce
`<expression>`		accept

Like the table-driven scanner you saw earlier, a bottom-up parser can be table-driven. The parse table encodes the production rules and the shift and reduce actions. Such a table is almost always generated by a compiler-compiler.

A table-driven parser can be very compact and extremely fast. However, for any significant grammar, the table can be nearly impossible to interpret manually. Error recovery can be especially tricky. A bottom-up parser can be very hard to debug if something goes wrong, such as if there is an error in the grammar.

Context-Free and Context-Sensitive Grammars

In a *context-free* grammar, each production rule has a single nonterminal symbol for its left side. For example:

```
<simple expression> ::= <term> + <term>
```

When a bottom-up parser matches the right side of a production rule on the parse stack, it can freely reduce the matching symbols to the terminal symbol at the left side of the rule. You saw this action several times above.

A language is said to be context-free if it is defined by a context-free grammar.

Context-free grammars are a subset of *context-sensitive* grammars. Context-sensitive grammars are more "powerful" in that they can define more languages than context-free grammars.

Production rules in a context-sensitive grammar may have the form

```
<A><B><C> ::= <A><b><C>
```

A parser can reduce the symbol `<b>` to `<B>` only in the context of symbols `<A>` and `<C>`.

For example, a context-sensitive grammar can express the language rule that an identifier must have been previously declared to be a variable before it can appear in an expression. In other words, the parser can reduce `<identifier>` to `<variable>` in an expression only in the context of a prior `<variable declaration>` for that identifier.

Context-sensitive grammars can be difficult for compiler writers to use. Therefore, you are likely better off using a context-free grammar and relying upon semantic actions such as building symbol tables to implement declaration rules, as you have done in this book.

Code Generation

This book took a simplistic approach to code generation: You did a postorder traversal of the parse tree and emitted code according to the tree nodes you visited. The approach did work; after all, the generated object code ran correctly. But as you may have surmised, there is more to code generation than just spitting out code that runs.

Instruction Selection

As a compiler writer, you must ask: What sequence of target machine instructions should the code generator emit? In previous chapters, you relied upon the parse trees and symbol tables to help answer this question, and you used code templates to record your answers.

As you designed the code generator, you had to make some decisions. For example, to load a constant value onto the runtime stack, you emitted a LDC, ICONST_*n*, or BIPUSH instruction. For the Pascal CASE statement, you emitted the LOOKUPSWITCH instruction, but for densely packed branch values, the TABLESWITCH instruction may have been better.

Consider the common Pascal assignment statement that increments an integer variable by 1. For example,

```
i := i + 1
```

If you assume that i is a local variable in slot #0 of the local variables array, one possible instruction sequence that you can emit is

```
iload_0
iconst_1
iadd
istore_0
```

But a shorter instruction sequence that would execute faster is

```
iinc 0 1
```

A *retargetable compiler* can generate code for multiple target machines and different instruction sets. You would need a set of code templates for each target machine. One way to build a retargetable compiler is to design it so that it uses the same symbol tables and parse trees for the different target machines, but you can swap different code generators in and out of the compiler back end.

Instruction Scheduling

With most target machine architectures, different instructions take different lengths of time to execute. For example, an instruction that multiplies two integer values together may take longer than an instruction that adds two integer values together. Instructions that operate on floating-point values generally take longer than the corresponding instructions that operate on integer values.

Instruction scheduling is a form of optimization to increase overall execution speed of a compiled program. You change the order that the code generator emits instructions, but without changing the program's semantics.

Here's a simple example. Suppose you are generating code for a target machine that can overlap instruction execution. In other words, the machine architecture supports *instruction-level parallelism*. Assume that the load and store instructions each takes 3 machines cycles to execute, the multiply instruction takes 2 cycles, and the add instruction takes 1 cycle.

The tables below show how the scheduled code takes advantage of instruction-level parallelism to save machine cycles over the original code.[2]

[2] Adapted from page 588 of *Engineering a Compiler*, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.

Original Code		
Cycle start	**Instruction**	**Operation**
1	`load`	$w \Rightarrow r1$
4	`add`	$r1 + r1 \Rightarrow r1$
5	`load`	$x \Rightarrow r2$
8	`mult`	$r1 * r2 \Rightarrow r1$
9	`load`	$y \Rightarrow r2$
12	`mult`	$r1 * r2 \Rightarrow r1$
13	`load`	$z \Rightarrow r2$
16	`mult`	$r1 * r2 \Rightarrow r1$
18	`store`	$r1 \Rightarrow w$

Scheduled Code		
Cycle start	**Instruction**	**Operation**
1	`load`	$w \Rightarrow r1$
2	`load`	$x \Rightarrow r2$
3	`load`	$y \Rightarrow r3$
4	`add`	$r1 + r1 \Rightarrow r1$
5	`mult`	$r1 * r2 \Rightarrow r1$
6	`load`	$z \Rightarrow r2$
7	`mult`	$r1 * r3 \Rightarrow r1$
9	`mult`	$r1 * r2 \Rightarrow r1$
11	`store`	$r1 \Rightarrow w$

The original code requires 20 cycles total and the scheduled code requires 13 cycles total. Both code sequences produce the same result. The scheduled code sequence does require a third register *r3* – a tradeoff that you must decide whether to make.

Register Allocation

You can think of the local variables array in each stack frame of the Java Virtual Machine as a set of registers. However, a real machine may have only a limited number of registers. These registers may also be categorized by how you can use them:

- General-purpose registers
- Floating-point registers
- Address registers

A smart code generator emits code that loads values into registers as often as possible, since performing operations on values in registers is faster than performing the operations on values in main memory. The code should also keep values in registers as long as possible. *Register allocation* is deciding which values should be in which registers.

A code generator assigns register on a per-routine invocation basis. During a procedure or function call, the generator should first emit instructions to save the caller's register contents. Later upon return from the routine, it emits instructions to restore the caller's register contents. A smart code generator only saves and restores the registers that a routine actually uses.

You face several challenges to doing register allocation correctly.

Because a machine has only a limited number of registers, you may need to to *spill* a register value into memory. To free up a register, the code generator emits code to store the register's contents into main memory. Later, it may need to emit code to reload that value from main memory back into the register.

If a pointer variable points to a value in main memory, then there is risk if you move that value into a register. If the value in the register changes, then the pointer variable will point to a value in memory that is *stale*.

Since registers are a limited resource, you want to use them as efficiently as possible. You should only keep the values of *live variables* in the registers. The compiler can perform *data-flow analysis* to determine which variables are live. You want to keep a variable's value in a register only if that variable will be used later during the program's execution.

Code Optimization

To be a master compiler writer, you want your code generator to generate better object code. To accomplish this feat, the back end must discover information about the runtime behavior of the source program. This can require several passes of the intermediate code before the code generator emits the object code. A back end compiler component called the *code optimizer* can modify the parse tree to allow the code generator to emit better code.

The generated object code is "better" if it:

- **Runs faster.** This is usually what compiler writers mean when they talk about code optimization.
- **Uses less memory.** This is especially important with machine architectures that have limited amounts of memory, such as embedded chips. Code that has a smaller "footprint" requires less storage and loads quicker into memory for execution.
- **Consumes less power.** Some machine instructions may consume more power than others. Power consumption is an important consideration with battery-powered laptops and devices such as cell phones and other hand-held computers.

There are major challenges to code optimization.

The primary challenge is safety. The code optimizer must not change the semantics of the source program. For example, the optimized compiled code must generate the same values as unoptimized code. You must never design a code optimizer that causes the code generator to emit code that computes the wrong values, only faster!

Good code optimization is difficult to achieve and it is time consuming; your compilations can take much longer. As a compiler writer, this is another tradeoff you must decide to make. Is generating optimized code worth the time and effort?

Debugging Compilers and Optimizing Compilers

You use a debugging, or development, compiler during program development. Such a compiler does fast compiles to give you fast turnaround. A development compiler may be coupled with an interpreter and an interactive source-level debugger.

After you believe that you've "thoroughly" debugged your program, you can use an optimizing, or *production*, compiler to generate optimized code. Thus, the code that you produce to ship to customers can be optimized.

Speed Optimizations

The following are brief descriptions of some common techniques to generate object code that is optimized for speed.

Constant Folding

Suppose a Pascal source program has the constant definition

```
CONST pi = 3.14;
```

Then for each appearance of the real expression `2*pi` in the program, the code optimizer alters the parse tree so that instead of emitting instructions to load the constant 2, load the constant 3.14, and multiply, the code generator simply emits a single instruction to load the constant 6.28.

Constant Propagation

The code optimizer analyzes a parse tree and determines that a variable `v` always has the value `c` within a set of source statements. It alters the parse tree so that when the code generator generates object code for those statements, the generator does not emit instructions to load the value of `v` from memory for each use of `v`. Instead, it emits an instruction to load the constant `c`.

Strength Reduction

Replace an operation by a faster equivalent operation. For example, suppose the source program contains the integer expression `5*i` in a tight loop. Multiplication is slower than addition, so the code optimizer modifies the parse tree to cause the code generator to emit code instead for `i + i + i + i + i`. Or, the code generator can emit code for `(4*i) + i` where the multiplication by 4 performed by an instruction that shifts the value of `i` two bits to the left.

Loop Unrolling

A looping statement has overhead. The code generator must emit code to initialize the loop control variable, test its value against a limit, and increment its value. This overhead can be expensive if the loop is itself inside of another loop.

For example, suppose you have the following set of loops in a Pascal program:

```
FOR i := 1 TO n DO BEGIN
    FOR j := 1 TO 3 DO BEGIN
        s[i,j] := a[i,j] + b[i,j]
    END
END
```

The code optimizer modifies the parse tree so that the code generator emits code as if the inner loop were "unrolled" into individual statements:

```
FOR i := 1 TO n DO BEGIN
    s[i,1] := a[i,1] + b[i,1];
    s[i,2] := a[i,2] + b[i,2];
    s[i,3] := a[i,3] + b[i,3];
```

END

Design Note

Of course, you can tell the application programmers to write their programs with unrolled loops, folded and propagated constants, strength reductions, etc. But a good policy is to tell programmers to concentrate on writing well-designed programs that work. Then let an optimizing compiler perform the optimizations to generate better code.

Dead Code Elimination

A sloppily written Pascal progam contains the `WHILE` loop

```
WHILE false DO
    BEGIN
        ...
    END
```

If there are no statement labels (and therefore the statements cannot be targets of `GOTO` statements), none of the statements in the compound statement can ever be executed. The code optimizer "prunes" such dead code from the parse tree, and therefore code generator simply does not emit any code forthe `WHILE` statement.

Common Subexpression Elimination

This is an obvious optimization, but it can be difficult for a code optimizer to do well. For example, suppose a Pascal program contains the assignment statement

```
x := y*(i - j*k) + (w + z/(i - j*k))
```

The code optimizer recognizes the common subexpression `i - j*k` and modifies the parse tree so that the code generator emits code as if the statement were instead

```
t := i - j*k;
x := y*t + (w + z/t);
```

where `t` is a temporary variable that the code optimizer created.
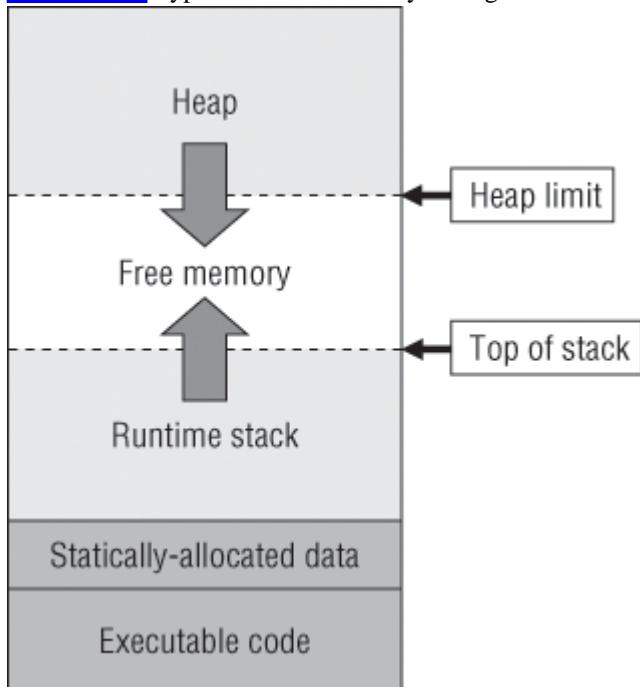
Runtime Memory Management

During run time, you can partition the target machine's memory conceptually into four areas:

1. Static memory, which contains the executable object code and statically-allocated data.

2. The **runtime stack,** which contains the activation records (or stack frames) which in turn contain the locally-scoped data.

3. The **heap,** which contains dynamically allocated data. For Pascal, these are data allocated at run time with `new`, and for Java, these are nonstatic objects.

4. Free memory, which is memory not currently in use.

Figure 19-4 shows how runtime memory can be managed.

Figure 19-4: Typical runtime memory management

Operating systems often include a *runtime memory manager* that provides memory management services to executing programs.

Heap and Stack

shows that the runtime stack grows in one direction as the executing program calls procedures and functions, and the heap grows in the opposite direction as the program allocates more dynamic data.

A *heap-stack collision* occurs if the runtime stack and the heap run into each other. This is an example of a runtime out-of-memory error.

Garbage Collection

A block of memory in the heap is deemed unused ("garbage") if there are no longer any references to it. An unused block should be deallocated ("garbage collected") so that it becomes available for use later by the executing program.

Java has automatic garbage collection. As the object program is executing, the runtime memory manager monitors the state of the heap. Periodically or whenever the heap becomes too full, the memory manager automatically performs garbage collection. Garbage collection can occur as a background task so the program continues to run (but perhaps more slowly), or the operating system may interrupt the program to allow the garbage collector to run and then resume executing the program.

Pascal and other languages like C do not have automatic garbage collection. An executing Pascal program executes `new` and a C program calls `malloc()` to allocate a data block in the heap. When the program no longer needs the data block, the Pascal program executes `dispose` and the C program calls `free()` to deallocate the block.

Automatic garbage collection frees you from worrying about managing the heap when you write application programs. However, automatic garbage collection may not be suitable in situations where your program must run as quickly as possible, or where your program's execution time must be predictable. It may not be acceptable for the garbage collector to interrupt the execution of your program or otherwise impact its performance.

Whether automatic or explicitly invoked by the executing program, garbage collection is important to keep the heap from growing uncontrollably and causing a heap-stack collision. Runtime memory managers can compact the heap and thereby shrink the heap's size after garbage collection.

Because automatic garbage collection executes asynchronously from the running program, it must rely upon various algorithms to determine whether or not a block of memory in the heap is still being used. Some common algorithms are described below.

Reference Counting

To enable *reference counting*, the runtime memory manager includes a counter value with each block of memory that it allocates in the heap. The counter starts at 0 and increments by 1 each time the executing program sets a pointer to the block. Whenever a pointer changes from pointing to the block to pointing elsewhere, the counter decrements by 1. If the counter reaches 0 again, the block becomes garbage and is eligible for deallocation.

A major problem with reference counting is circular references. Two blocks of memory that would otherwise be garbage each contains a pointer to the other block. Their reference counts never reach 0.

Mark and Sweep

To perform the mark and sweep garbage collection algorithm, the runtime memory manager makes a pass over the heap to *mark* (record) all the allocated blocks of memory that it can reach via pointers. There are various marking techniques. The memory manager makes a second pass to *sweep* (deallocate) memory blocks that it did not previously mark.

Stop and Copy

To implement the stop and copy garbage collection algorithm, the runtime memory manager divides the heap into halves. It allocates memory in only one half at a time.

When the half of the heap currently in use becomes too full, the memory manager stops allocating new memory blocks. It copies all the allocated blocks from their half of the heap to the other half, thereby compacting the allocated blocks. Memory allocation then resumes in this other half of the heap.

Compiling Object-Oriented Languages

The Pascal subset you used in previous chapters as the source language is procedure-oriented but not object-oriented.[3] Writing compilers for object-oriented languages has extra challenges that are briefly described below.

[3] The Jasmin assembler took care of this for you.

Method Overloading and Name Mangling

Object-oriented languages can support *method overloading*. Two or more methods in the same class can have the same name but different numbers or types of formal parameters. For example,

```
void swap(Integer i, Integer j) { … }
void swap(Float x, Float y) { … }
```

To distinguish the two methods in the compiled code, compilers such as the C++ compiler do *name mangling*. The emitted code encodes the parameter number and types and the return type in a generated method name. For example,

```
swap%void%Integer%Integer
swap%void%Float%Float
```

The Java Virtual Machine generally doesn't need name mangling. As you've seen in the previous chapters, method calls in Jasmin explicitly specify the parameter number and types and the return type. For example,

```
invokevirtual swap (Ljava/lang/Integer;Ljava/lang/Integer;)V
```

However, the JVM requires name mangling for inner classes, such as `Outer$Inner.class` for a class `Outer` that has an inner class `Inner`.
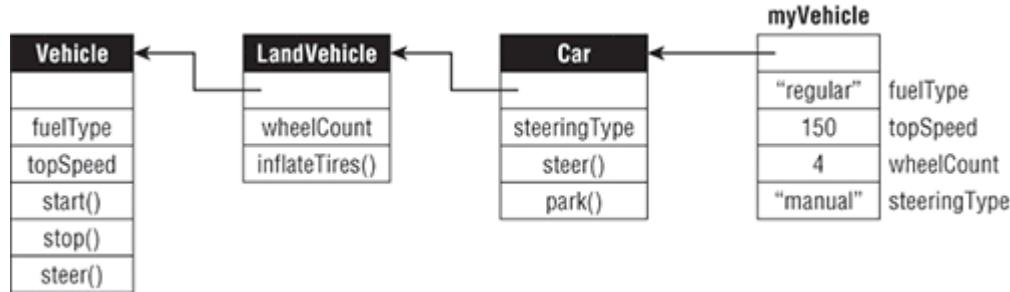
Inheritance

A compiler generates *class objects* that contain information about the source program's classes.[4] Each class object has a reference to its superclass object. Each allocated instance object has a reference to its class object. Figure 19-5 shows an example an inheritance chain.

[4] Later versions of Pascal were object-oriented.

Figure 19-5: An inheritance chain

An *instance object* object contains only data and inherits all the methods and nonstatic fields from its superclasses.

Virtual Methods

The type of an instance object may not be known until run time. Therefore, whenever a method is invoked on the object, such as `vehicle.start()` in Figure 19-5, a search up the inheritance chain may be needed to determine which method to execute. This search can be made more efficiently if each class object includes a *virtual dispatch table*, or *vtable*. This table contains references to all the inherited methods.

The JVM's INVOKEVIRTUAL instruction calls a method on its instance object operand which is currently on top of the operand stack. For example,

```
invokevirtual java/io/PrintStream/println (Ljava/lang/String;)V
```

The JVM looks at the runtime type of the instance object to determine which method to call.

Compiler-Compilers

A *compiler-compiler* is a software tool that generates compilers. You feed it a precise description of the source language (the language's grammar) and it outputs key parts of a compiler, generally the scanner and parser. Compiler-compilers can generate scanners and parsers in a high-level language, such as Java or C, so you can read the code and modify it if necessary.

The rest of this chapter briefly describes several popular compiler-compiler tools: JavaCC and Yacc and Lex.

JavaCC

The JavaCC compiler-compiler generates scanners and parsers written in Java. It can also generate Java code to build parse trees and walk over the nodes.

You supply JavaCC with the grammar of the source language for which you want JavaCC to generate a scanner and a parser. (The documentation for JavaCC calls the scanner a *tokenizer*.) This grammar must be in a special format that contains regular expressions that describe the tokens and Extended BNF statements that describe the syntax rules. JavaCC uses the regular expressions to generate a scanner and the EBNF statements to generate a parser. Both the generated scanner and parser will be written in Java.[5]

[5] Java code that is generated by a tool can be hard to read.

[6] Adapted from page 122 of *Generating Parsers with JavaCC*, by Tom Copeland, Centennial Books, 2007.

Suppose you want JavaCC to generate a scanner and a parser to recognize a simple expression consisting of a number followed by a plus sign followed by another number, such as `23+456`. A number consists of one or more digits.

Listing 19-2 shows the grammar file `calculator_parser.jj`. The grammar file combines Java code, regular expressions, and EBNF.[6]

Listing 19-2: The simple JavaCC grammar file `calculator_parser.jj`

```
PARSER_BEGIN(Calculator)
import java.io.*;
```

```
public class Calculator
{
    public static void main(String[] args)
    {
        Reader sr = new StringReader(args[0]);
        Calculator calc = new Calculator(sr);

        try {
            calc.Expression();
        }
        catch (ParseException ex) {
            ex.printStackTrace();
        }
    }
}
PARSER_END(Calculator)

TOKEN : {
    <DIGITS : (["0"-"9"])+>
  | <PLUS : "+">
}

void Expression() : {}
{
    {System.out.println("Expression starts");}
    Operator()
    {System.out.println("Expression ends");}
}

void Operator() : {}
{
    Operand()
    <PLUS> {System.out.println("Operator: " + tokenImage[PLUS]);}
    Operand()
}

void Operand() : {Token t;}
{
    t=<DIGITS> {System.out.println("Operand: " + t.image);}
}
```

Between `PARSER_BEGIN` and `PARSER_END` is straightforward Java code that is the main method. The `TOKEN` section contains the regular expressions for the `DIGITS` and `PLUS` tokens.

The grammar rules follow the `TOKEN` section. Write each production rule like a method. The name of the method is the name of the nonterminal symbol, and the body of the method is the rule. So each "method" is the JavaCC way to write an EBNF rule. Within each rule, you can insert Java statements within curly braces { and }.

Invoke JavaCC on this grammar file with a command line similar to

```
javacc calculator_parser.jj
```

JavaCC generates a small number of Java files that implement the scanner and the parser. These files incorporate the Java code from the grammar file. The generated parser is a top-down parser.

Compile the generated Java files with a command line similar to

```
javac *.java
```

Run the calculator parser with a command line similar to

```
java Calculator 23+456
```

The output will be

```
Expression starts
Operand: 23
Operator: "+"
Operand: 456
Expression ends
```

The Java code embedded in a production rule of the grammar file executes whenever the parser applies that rule.

Lex and Yacc

Lex and Yacc are the classic compiler-compiler tools that are standard on many Unix and Linux systems.[7]

[7] The GNU versions of these tools are Flex and Bison, respectively.

Lex generates a scanner written in C, and Yacc ("Yet another compiler-compiler") generates a parser written in C. The generated parser is a bottom-up shift-reduce parser.

Like JavaCC, these tools require input files that specify the grammar of the source language. Lex reads a file containing token definitions, and Yacc reads an input file containing production rules.

For example, Listing 19-3 shows input file `calc.l` for Lex.

Listing 19-3: The example input file `calc.l` for Lex

```
%{
#include "calc.tab.h"
extern lineno;
%}
%option noyywrap

%%

[ \t]                  {;}  /* skip white space */
[0-9]+\.?|[0-9]*\.[0-9]+ {sscanf(yytext,"%lf",&yylval); return NUMBER;}
\n                     {lineno++ ;return &apos;\n&apos;;}
.                      {return yytext[0];}  /* everything else */
```

Listing 19-4 shows input file `calc.y` for Yacc.

Listing 19-4: The example input file `calc.y` for Yacc

```
%{
#define YYSTYPE double /* data type of yacc stack */
%}

%token NUMBER
%left '+' '-' /* left associative, same precedence */
%left '*' '/' /* left associative, higher precedence */

%%

exprlist: /* nothing */
    | exprlist '\n'
    | exprlist expr '\n' {printf("\t%lf\n", $2);}
    ;

expr: NUMBER        {$$ = $1;}
    | expr '+' expr {$$ = $1+$3;}
    | expr '-' expr {$$ = $1-$3;}
    | expr '*' expr {$$ = $1*$3;}
    | expr '/' expr {$$ = $1/$3;}
    | '(' expr ')'  {$$ = $2;}
    ;

%%

#include <stdio.h>
#include <ctype.h>

char *progname; /* for error message */
int lineno=1;

main(int argc, char *argv[])
{
    progname=argv[0];
    yyparse();
}

yyerror(char *s) /* called for yacc syntax error */
{
    warning(s, (char *) 0);
}

warning(char *s, char *t) /* print warning message */
{
    fprintf(stderr, "%s: %s",progname,s);
    if (t) fprintf(stderr," %s", t);
    fprintf(stderr," near line %d\n",lineno);
}
```

Both `calc.l` and `calc.y` contain C code that will be embedded in the generated scanner and parser.

This concludes a very brief overview of compiler-writing topics not covered in the previous chapters. Yet there is still much more out there! Invent the ultimate programming language and then write the best possible compiler for it.

Index

A

E

S