CHAPTER

1

Introduction

COPYRIGHTED MATERIAL

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).
- 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.

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

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]

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

[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.

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.

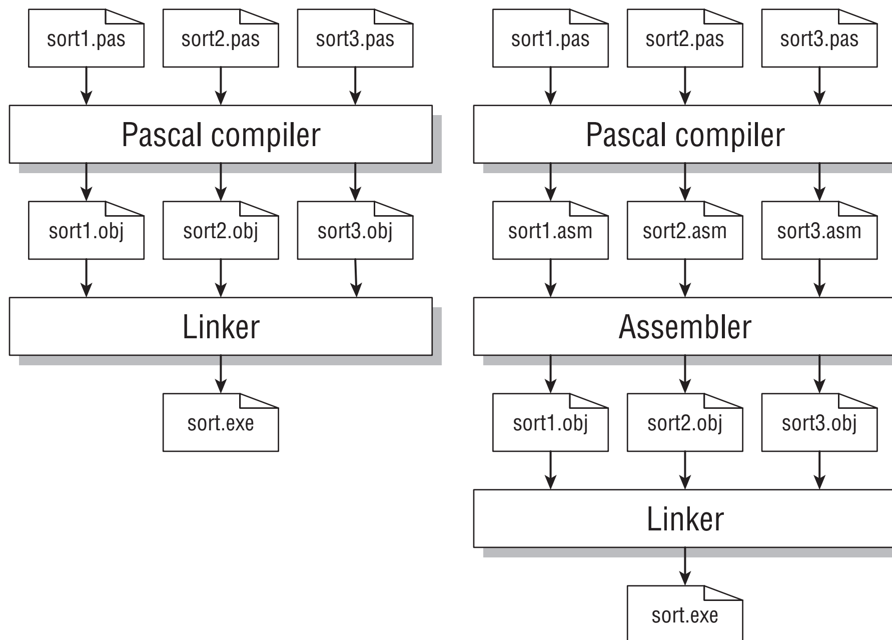

Figure 1-1: The diagram on the left shows a compiler translating a Pascal program consisting of three source files `sort1.pas`, `sort2.pas`, and `sort3.pas` into three corresponding machine-language object files `sort1.obj`, `sort2.obj`, and `sort3.obj`. The linker then combines the object files (along 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.

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]

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.

[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.

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]

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.

[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.

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.

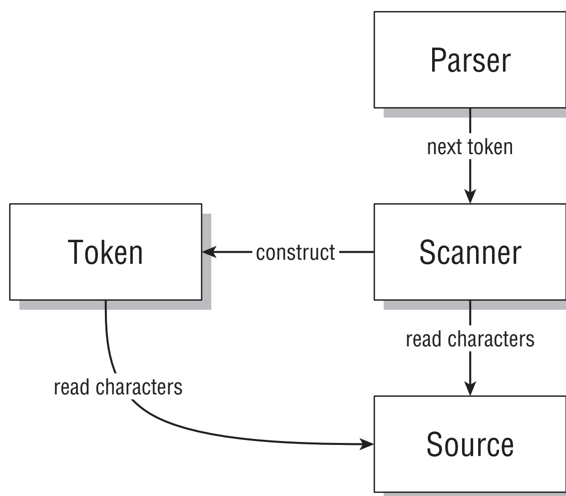

Figure 1-2: The conceptual design for the front end

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.

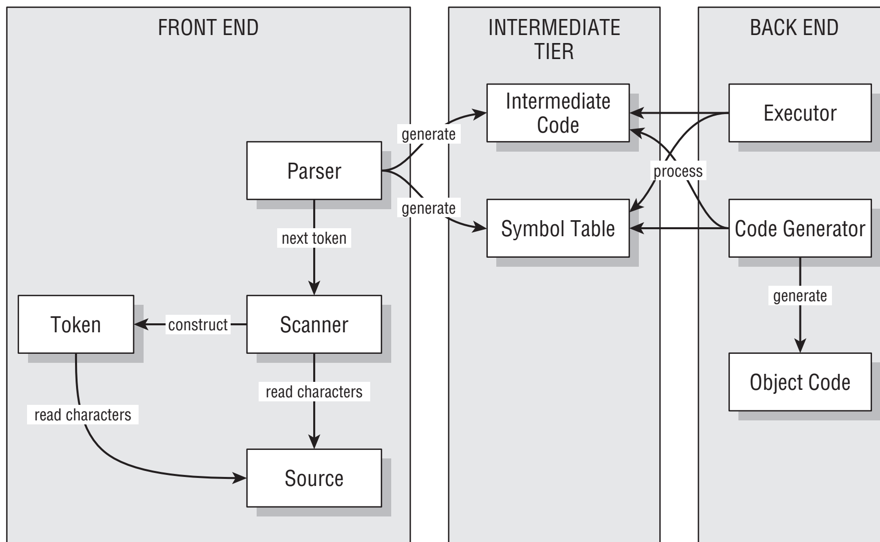

Figure 1-3: A more complete conceptual design

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.