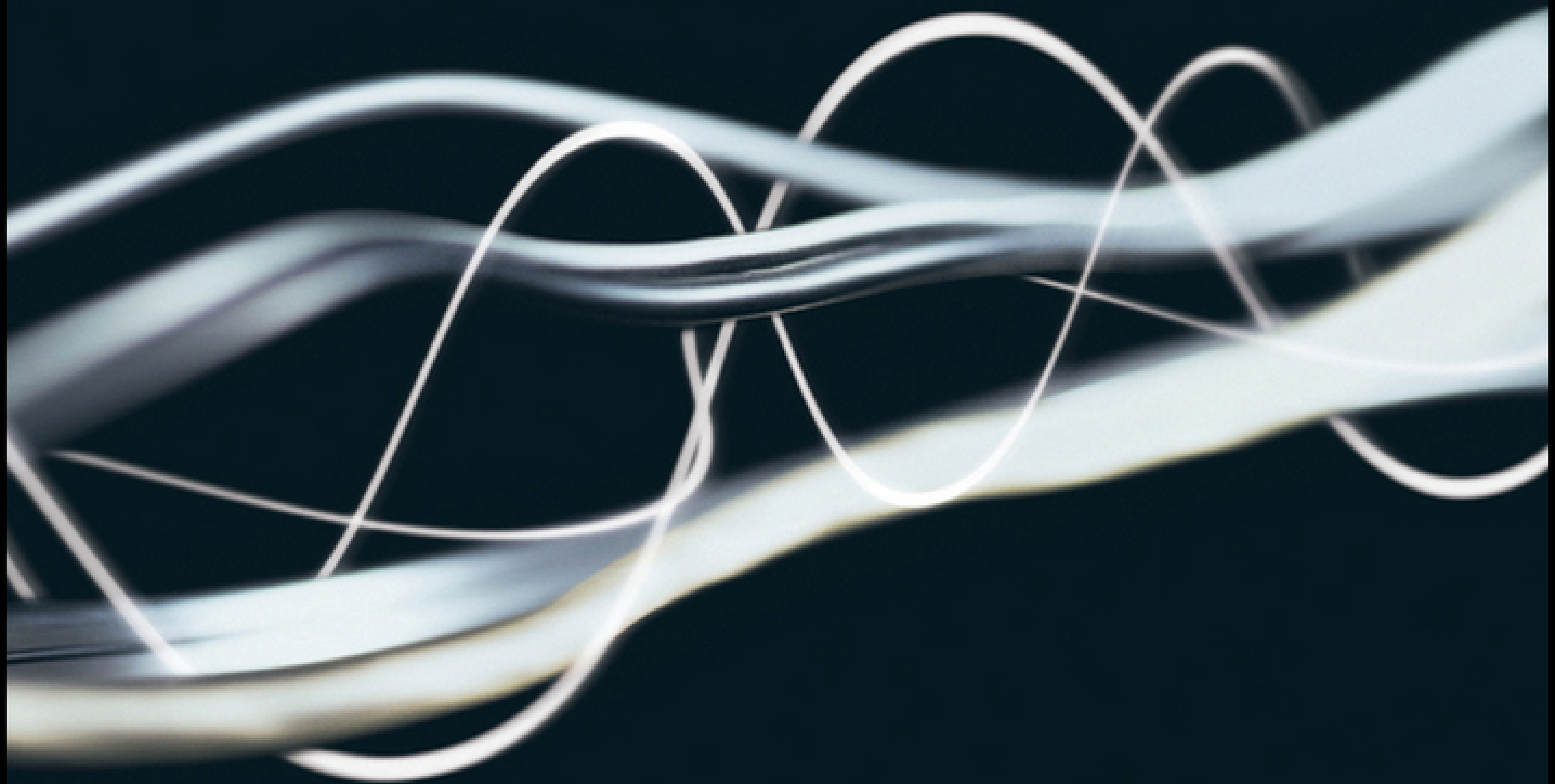
THIRD EDITION

Writing Compilers and Interpreters
A Software Engineering Approach



Ronald Mak

Table of Contents

Writing Compilers and Interpreters

A Modern Software Engineering Approach Using Java®

Third Edition

Ronald Mak

WILEY
Wiley Publishing, Inc.

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

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-17707-5

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Control Number: 2009931754

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

Executive Editor
Carol Long

Project Editor
Tom Dinse

Technical Editor
Chris Tseng

Production Editor
Daniel Scribner

Copy Editor
Christopher Jones

Editorial Director
Robyn B. Siesky

Editorial Manager
Mary Beth Wakefield

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Barry Pruett

Associate Publisher
Jim Minatel

Project Coordinator, Cover
Lynsey Stanford

Compositor
Craig J. Woods, Happenstance Type-O-Rama

Proofreader
Publication Services, Inc.

Indexer
Ron Strauss

Cover Image
© Punchstock

Acknowledgments

Carol Long, Executive Acquisitions Editor at Wiley, first contacted me a couple of years ago to persuade me to do this third edition. She stayed on top of my efforts to complete it.

Tom Dinse, Senior Development Editor, did a great job making sure I uploaded my chapter manuscripts on time and coordinated all the editing work. Tom was a pleasure to work with.

My students of CS 153, Concepts of Compiler Design, in the Department of Computer Science at San José State University (California) used early manuscripts of the chapters and pointed out problems to me. I realize most of them may never write a compiler during their careers, but I hope they benefited from learning how to work together in small programming teams to design and develop some very complex software.

Thanks to Phil Edwards, my former colleague at the Lawrence Livermore National Laboratory, who read several of the early chapters and helped me clear up some confusing statements.

Finally, special thanks to my technical editor, computer science professor Dr. Chris Tseng at San José State University. He read all the chapters, ran all the programs, and his suggestions and comments helped me improve the text.

Large, complex programs such as compilers and interpreters are challenging to get right. The

programs in this book are for educational purposes only and are not production quality. I accept the blame for any bugs they contain. However, neither I nor Wiley Publishing can be responsible for any damages these programs may cause if you use them for any other purposes.

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.

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.