Programming Language Processors in Java

COMPILERS AND INTERPRETERS

DAVID A WATT
University of Glasgow, Scotland
and
DERYCK F BROWN
The Robert Gordon University, Scotland

An imprint of Pearson Education
Harlow, England · London · New York · Reading, Massachusetts · San Francisco · Toronto · Don Mills, Ontario · Sydney
Tokyo · Singapore · Hong Kong · Seoul · Taipei · Cape Town · Madrid · Mexico City · Amsterdam · Munich · Paris · Milan

Pearson Education Limited
Edinburgh Gate
Harlow
Essex, CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
http://www.pearsoneduc.com

First published 2000

ISBN 0 130 25786 9

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

Library of Congress Cataloging-in-Publication Data
Watt, David A. (David Anthony)
Programming language processors in Java : compilers and interpreters / David A. Watt and Deryck F. Brown
p. cm.
Includes bibliographical references.
ISBN 0–13–025786–9 (case)
1. Java (Computer program language) 2. Compilers (Computer programs) 3. Interpreters (Computer programs) I. Brown, Deryck F. II. Title.

QA76.73.J38 W385 1999
005.4´53–dc21 99–050395

10 9 8 7 6 5 4 3 2
04 03 02 01 00

Typeset by 7
Printed an bound in Great Britain by Biddles Ltd, *www.biddles.co.uk*

Contents

Trademark notice

The following are trademarks or registered trademarks of their respective companies: Ada is a trademark of the US Department of Defense – Ada Joint Program Office; Intel 80386 and Pentium are trademarks of Intel Corporation; Java, JavaCC, and JDK are trademarks of Sun Microsystems, Inc.; JBuilder is a trademark of Borland International, Inc.; PowerPC is a trademark of International Business Machines Corporation; SPARC is a trademark of SPARC International, Inc.; UNIX is a trademark licensed through X/Open Company Ltd.

Preface

The subject of this book is the implementation of programming languages. Programming language processors are programs that process other programs. The primary examples of language processors are compilers and interpreters.

Programming languages are of central importance in computer science. They are the most fundamental tools of software engineers, who are completely dependent on the quality of the language processors they use. There is an interplay between the design of programming languages and computer instruction sets: compilers must bridge the gap between high-level languages and machine code. And programming language design itself raises strong feelings among computer scientists, as witnessed by the proliferation of language paradigms. Imperative and object-oriented languages are currently dominant in terms of actual usage, and it is on the implementation of such languages that this book focuses.

Programming language implementation is a particularly fascinating topic, in our view, because of its close interplay between theory and practice. Ever since the dawn of computer science, the engineering of language processors has driven, and has been vastly improved by, the development of relevant theories.

Nowadays, the principles of programming language implementation are very well understood. An experienced compiler writer can implement a simple programming language about as fast as he or she can type. The basic techniques are simple yet effective, and can be lucidly presented to students. Once the techniques have been mastered, building a compiler from scratch is essentially an exercise in software engineering.

A textbook example of a compiler is often the first complete program of its size seen by computer science students. Such an example should therefore be an exemplar of good software engineering principles. Regrettably, many compiler textbooks offend these principles. This textbook, based on a total of about twenty-five years' experience of teaching programming language implementation, aims to exemplify good software engineering principles at the same time as explaining the specific techniques needed to build compilers and interpreters.

The book shows how to design and build simple compilers and interpreters using the object-oriented programming language Java. The reasons for this choice are two-fold. First, object-oriented methods have emerged as a dominant software engineering technology, yielding substantial improvements in software modularity, maintainability,

and reusability. Secondly, Java itself has experienced a prodigious growth in popularity since its appearance as recently as 1994, and that for good technical reasons: Java is simple, consistent, portable, and equipped with an extremely rich class library. Soon we can expect all computer science students to have at least some familiarity with Java.

A programming languages series

This is the fourth of a series of books on programming languages:

- *Programming Language Concepts and Paradigms*
- *Programming Language Syntax and Semantics*
- *Programming Language Processors*
- *Programming Language Processors in Java*

Programming Language Concepts and Paradigms studies the concepts underlying programming languages, and the major language paradigms that use these concepts in different ways; in other words, it is about language design. *Programming Language Syntax and Semantics* shows how we can formally specify the syntax (form) and semantics (meaning) of programming languages. *Programming Language Processors* studies the implementation of programming languages, examining language processors such as compilers and interpreters, and using Pascal as the implementation language. *Programming Language Processors in Java* likewise studies the implementation of programming languages, but now using Java as the implementation language and object-oriented design as the engineering principle; moreover, it introduces basic techniques for implementing object-oriented languages.

This series attempts something that has not previously been achieved, as far as we know: a broad study of all aspects of programming languages, using consistent terminology, and emphasizing connections likely to be missed by books that deal with these aspects separately. For example, the concepts incorporated in a language must be defined precisely in the language's semantic specification. Conversely, a study of semantics helps us to discover and refine elegant and powerful new concepts, which can be incorporated in future language designs. A language's syntax underlies analysis of source programs by language processors; its semantics underlies object code generation and interpretation. Implementation is an important consideration for the language designer, since a language that cannot be implemented with acceptable efficiency will not be used.

The books may be read as a series, but each book is sufficiently self-contained to be read on its own, if the reader prefers.

Content of this book

Chapter 1 introduces the topic of the book. It reviews the concepts of high-level programming languages, and their syntax, contextual constraints, and semantics. It explains what a language processor is, with examples from well-known programming systems.

Chapter 2 introduces the basic terminology of language processors: translators, compilers, interpreters, source and target languages, and real and abstract machines. It goes on to study interesting ways of using language processors: interpretive compilers, portable compilers, and bootstrapping. In this chapter we view language processors as 'black boxes'. In the following chapters we look inside these black boxes.

Chapter 3 looks inside compilers. It shows how compilation can be decomposed into three principal phases: syntactic analysis, contextual analysis, and code generation. It also compares different ways of designing compilers, leading to one-pass and multi-pass compilation.

Chapter 4 studies syntactic analysis in detail. It decomposes syntactic analysis into scanning, parsing, and abstract syntax tree construction. It introduces recursive-descent parsing, and shows how a parser and scanner can be systematically constructed from the source language's syntactic specification.

Chapter 5 studies contextual analysis in detail, assuming that the source language exhibits static bindings and is statically typed. The main topics are identification, which is related to the language's scope rules, and type checking, which is related to the language's type rules.

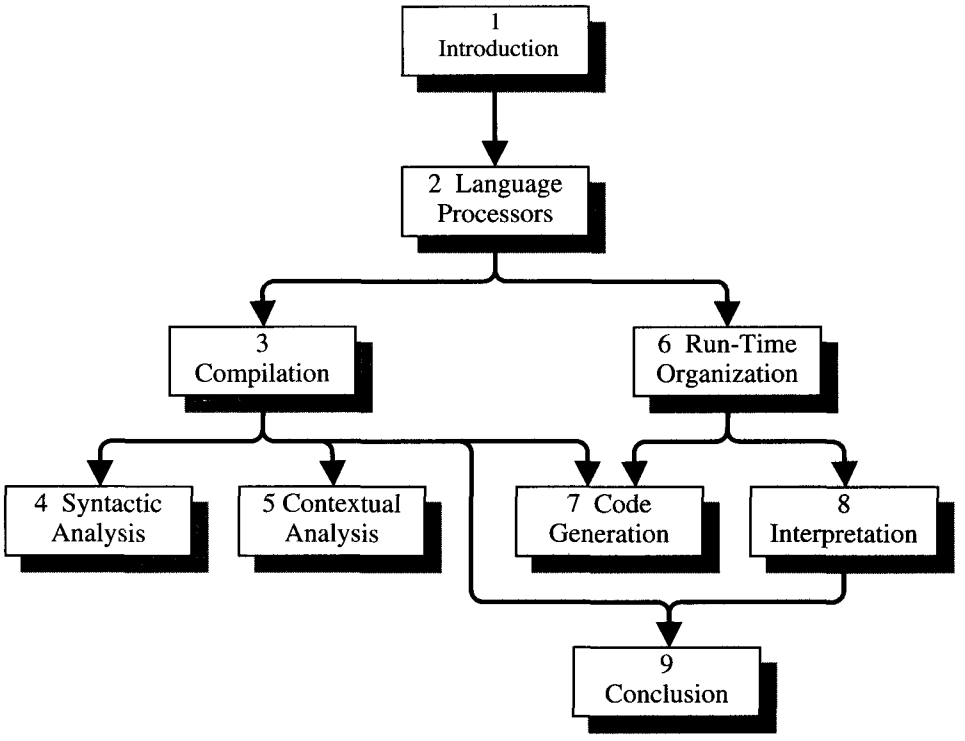
Chapter 6 prepares for code generation by discussing the relationship between the source language and the target machine. It shows how target machine instructions and storage must be marshaled to support the higher-level concepts of the source language. The topics covered include data representation, expression evaluation, storage allocation, routines and their arguments, garbage collection, and the run-time organization of simple object-oriented languages.

Chapter 7 studies code generation in detail. It shows how to organize the translation from source language to object code. It relates the selection of object code to the semantics of the source language. As this is an introductory textbook, only code generation for a stack-based target machine is covered. (The more difficult topics of code generation for a register-based machine, and code transformations are left to more advanced textbooks.)

Chapter 8 looks inside interpreters. It gives examples of interpreters for both low-level and high-level languages.

Chapter 9 concludes the book. It places the implementation of a programming language in the context of the language's life cycle, along with design and specification. It also discusses quality issues, namely error reporting and efficiency.

There are several possible orders for studying the main topics of this book. The chapter on interpretation can be read independently of the chapters on compilation. Within the latter, the chapters on syntactic analysis, contextual analysis, and code generation can be read in any order. The following diagram summarizes the dependencies between chapters.



Examples and case studies

The methods described in this textbook are freely illustrated by examples. In Chapter 2, the examples are of language processors for real programming languages. In the remaining chapters, most examples are based on smaller languages, in order that the essential points can be conveyed without the reader getting lost in detail.

A complete programming language is a synthesis of numerous concepts, which often interact with one another in quite complicated ways. It is important that the reader understands how we cope with these complications in implementing a complete programming language. For this purpose we use the programming language Triangle as a case study. An overview of Triangle is given in Section 1.4. A reader already familiar with a Pascal-like language should have no trouble in reading Triangle programs. A complete specification of Triangle is given in Appendix B; this includes a formal specification of its syntax, but is otherwise informal.

We designed Triangle for two specific purposes: to illustrate how a programming language can be formally specified (in the companion textbook *Programming Language Syntax and Semantics*), and to illustrate how a programming language can be implemented. Ideally we would use a real programming language, such as Pascal or Java, for these purposes. In practice, however, real languages are excessively complicated. They contain many features that are tedious but unilluminating to specify and to implement.

Although Triangle is a model language, it is rich enough to write interesting programs and to illustrate basic methods of specification and implementation. Finally, it can readily be extended in various ways (such as adding new types, new control structures, or packages), and such extensions are a basis for a variety of projects.

Educational software

A Triangle language processor is available *for educational use* in conjunction with this textbook. The Triangle language processor consists of: a compiler for Triangle, which generates code for TAM (Triangle Abstract Machine); an interpreter for TAM; and a disassembler for TAM. The tools are written entirely in Java, and will run on any computer equipped with a JVM (Java Virtual Machine). You can download the Triangle language processor from our Web site:

```
www.dcs.gla.ac.uk/~daw/books/PLPJ/
```

Exercises and projects

Each chapter of this book is followed by a number of relevant exercises. These vary from short exercises, through longer ones (marked *), up to truly demanding ones (marked **) that could be treated as projects.

A typical exercise is to apply the methods of the chapter to a very small toy language, or a minor extension of Triangle.

A typical project is to implement some substantial extension to Triangle. Most of the projects are gathered together at the end of Chapter 9; they require modifications to several parts of the Triangle compiler, and should be undertaken only after reading up to Chapter 7 at least.

Readership

This book and its companions are aimed at junior, senior, and graduate students of computer science and information technology, all of whom need some understanding of the fundamentals of programming languages. The books should also be of interest to professional software engineers, especially project leaders responsible for language evaluation and selection, designers and implementors of language processors, and designers of new languages and extensions to existing languages.

The basic prerequisites for this textbook are courses in programming and data structures, and a course in programming languages that covers at least basic language concepts and syntax. The reader should be familiar with Java, and preferably at least one other high-level language, since in studying implementation of programming languages it is important not to be unduly influenced by the idiosyncrasies of a particular language. All the algorithms in this textbook are expressed in Java.

The ability to read a programming language specification critically is an essential skill. A programming language implementor is forced to explore the entire language, including its darker corners. (The ordinary programmer is wise to avoid these dark

corners!) The reader of this textbook will need a good knowledge of syntax, and ideally some knowledge of semantics; these topics are briefly reviewed in Chapter 1 for the benefit of readers who might lack such knowledge. Familiarity with BNF and EBNF (which are commonly used in language specifications) is essential, because in Chapter 4 we show how to exploit them in syntactic analysis. No knowledge of formal semantics is assumed.

The reader should be comfortable with some elementary concepts from discrete mathematics – sets and recursive functions – as these help to sharpen understanding of, for example, parsing algorithms. Discrete mathematics is essential for a deeper understanding of compiler theory; however, only a minimum of compiler theory is presented in this book.

This book and its companions attempt to cover all the most important aspects of a large subject. Where necessary, depth has been sacrificed for breadth. Thus the really serious student will need to follow up with more advanced studies. Each book has an extensive bibliography, and each chapter closes with pointers to further reading on the topics covered by the chapter.

Acknowledgments

Most of the methods described in this textbook have long since passed into compiler folklore, and are almost impossible to attribute to individuals. Instead, we shall mention people who have particularly influenced us personally.

For providing a stimulating environment in which to think about programming language issues, we are grateful to colleagues in the Department of Computing Science at the University of Glasgow, in particular Malcolm Atkinson, Muffy Calder, Quintin Cutts, Peter Dickman, Bill Findlay, John Hughes, John Launchbury, Hermano Moura, John Patterson, Simon Peyton Jones, Fermin Reig, Phil Trinder, and Phil Wadler. We have also been strongly influenced, in many different ways, by the work of Peter Buneman, Luca Cardelli, Edsger Dijkstra, Jim Gosling, Susan Graham, Tony Hoare, Jean Ichbiah, Mehdi Jazayeri, Robin Milner, Peter Mosses, Atsushi Ohori, Bob Tennent, Jim Welsh, and Niklaus Wirth.

We wish to thank the reviewers for reading and providing valuable comments on an earlier draft of this book. Numerous cohorts of undergraduate students taking the *Programming Languages 3* module at the University of Glasgow made an involuntary but essential contribution by class-testing the Triangle language processor, as have three cohorts of students taking the *Compilers* module at the Robert Gordon University.

We are particularly grateful to Tony Hoare, editor of the Prentice Hall International Series in Computer Science, for his encouragement and advice, freely and generously offered when these books were still at the planning stage. If this book is more than just another compiler textbook, that is partly due to his suggestion to emphasize the connections between compilation, interpretation, and semantics.

Glasgow and Aberdeen D.A.W.
July, 1999 D.F.B.