

教育部 高等教育司 推荐
国外优秀信息科学与技术系列教学用书

现代编译程序实现

—— Java 语言

(第二版 影印版)

**MODERN COMPILER
IMPLEMENTATION IN JAVA**
(Second Edition)

■ Andrew W. Appel
with Jens Palsberg



高等教育出版社
Higher Education Press

教育部高等教育司推荐
国外优秀信息科学与技术系列教学用书

现代编译程序实现

——Java 语言

(第二版 影印版)

MODERN COMPILER
IMPLEMENTATION

(Second Edition)

Andrew W. Appel
with Jens Palsberg

江苏工业学院图书馆
藏书章



高等教育出版社

图字: 01-2003-1076 号

Modern Compiler Implementation in Java, Second Edition

Andrew W. Appel, with Jens Palsberg

本版本仅获准在中华人民共和国大陆地区发行和销售(不包括香港、台湾和澳门以及其他地区)。

Originally published by Cambridge University Press in 2002.

This reprint edition is published with the permission of the Syndicate of the Press of the University of Cambridge, Cambridge, England.

原版由剑桥大学出版社于 2002 年出版。

本影印版由英国剑桥的剑桥大学出版集团授权影印。

Modern Compiler Implementation in Java, 2e by Andrew W. Appel, with Jens Palsberg, Copyright © Cambridge University Press 2002

This book is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

图书在版编目(CIP)数据

现代编译程序实现: Java 语言: 第 2 版 / (美)阿佩尔 (Appel, A. W.), (美)帕尔斯伯格 (Palsberg, J.) 编著. —影印本. —北京: 高等教育出版社, 2003.

书名原文: Modern Compiler Implementation in Java

ISBN 7-04-013501-9

I. 现... II. ①阿... ②帕... III. ①编译程序—程序设计—高等学校—教材—英文 ②JAVA 语言—程序设计—高等学校—教材—英文 IV. TP314

中国版本图书馆 CIP 数据核字 (2003) 第 069930 号

出版发行	高等教育出版社	购书热线	010-64054588
社 址	北京市西城区德外大街 4 号	免费咨询	800-810-0598
邮政编码	100011	网 址	http://www.hep.edu.cn
总 机	010-82028899		http://www.hep.com.cn
经 销	新华书店北京发行所		
印 刷	北京民族印刷厂		
开 本	787 × 1092 1/16	版 次	2003 年 8 月第 1 版
印 张	32	印 次	2003 年 8 月第 1 次印刷
字 数	610 000	定 价	38.00 元

本书如有缺页、倒页、脱页等质量问题, 请到所购图书销售部门联系调换。

版权所有 侵权必究

前 言

20 世纪末, 以计算机和通信技术为代表的信息科学和技术对世界经济、科技、军事、教育和文化等产生了深刻影响。信息科学技术的迅速普及和应用, 带动了世界范围信息产业的蓬勃发展, 为许多国家带来了丰厚的回报。

进入 21 世纪, 尤其随着我国加入 WTO, 信息产业的国际竞争将更加激烈。我国信息产业虽然在 20 世纪末取得了迅猛发展, 但与发达国家相比, 甚至与印度、爱尔兰等国家相比, 还有很大差距。国家信息化的发展速度和信息产业的国际竞争能力, 最终都将取决于信息科学技术人才的质量和数量。引进国外信息科学和技术优秀教材, 在有条件的学校推动开展英语授课或双语教学, 是教育部为加快培养大批高质量的信息技术人才采取的一项重要举措。

为此, 教育部要求由高等教育出版社首先开展信息科学和技术教材的引进试点工作。同时提出了两点要求, 一是要高水平, 二是要低价格。在高等教育出版社和信息科学技术引进教材专家组的努力下, 经过比较短的时间, 第一批引进的 20 多种教材已经陆续出版。这套教材出版后受到了广泛的好评, 其中有不少是世界信息科学技术领域著名专家、教授的经典之作和反映信息科学技术最新进展的优秀作品, 代表了目前世界信息科学技术教育的一流水平, 而且价格也是最优惠的, 与国内同类自编教材相当。

这项教材引进工作是在教育部高等教育司和高教社的共同组织下, 由国内信息科学技术领域的专家、教授广泛参与, 在对大量国外教材进行多次遴选的基础上, 参考了国内和国外著名大学相关专业的课程设置进行系统引进的。其中, John Wiley 公司出版的贝尔实验室信息科学研究中心副总裁 Silberschatz 教授的经典著作《操作系统概念》, 是我们经过反复谈判, 做了很多努力才得以引进的。William Stallings 先生曾编写了在美国深受欢迎的信息科学技术系列教材, 其中有多种教材获得过美国教材和学术著作者协会颁发的计算机科学与工程教材奖, 这批引进教材中就有他的两本著作。留美中国学者 Jiawei Han 先生的《数据挖掘》是该领域中具有里程碑意义的著作。由达特茅斯学院 Thomas Cormen 和麻省理工学院、哥伦比亚大学的几

位学者共同编著的经典著作《算法导论》，在经历了 11 年的锤炼之后于 2001 年出版了第二版。目前任教于美国 Massachusetts 大学的 James Kurose 教授，曾在美国三所高校先后 10 次获得杰出教师或杰出教学奖，由他主编的《计算机网络》出版后，以其体系新颖、内容先进而倍受欢迎。在努力降低引进教材售价方面，高等教育出版社做了大量和细致的工作。这套引进的教材体现了权威性、系统性、先进性和经济性等特点。

教育部也希望国内和国外的出版商积极参与此项工作，共同促进中国信息技术教育和信息产业的发展。我们在与外商的谈判工作中，不仅要坚定不移地引进国外最优秀的教材，而且还要千方百计地将版权转让费降下来，要让引进教材的价格与国内自编教材相当，让广大教师和学生负担得起。中国的教育市场巨大，外国出版公司和国内出版社要通过扩大发行数量取得效益。

在引进教材的同时，我们还应做好消化吸收，注意学习国外先进的教学思想和教学方法，提高自编教材的水平，使我们的教学和教材在内容体系上，在理论与实践的结合上，在培养学生的动手能力上能有较大的突破和创新。

目前，教育部正在全国 35 所高校推动示范性软件学院的建设和实施，这也是加快培养信息科学技术人才的重要举措之一。示范性软件学院要立足于培养具有国际竞争力的实用性软件人才，与国外知名高校或著名企业合作办学，以国内外著名 IT 企业为实践教学基地，聘请国内外知名教授和软件专家授课，还要率先使用引进教材开展教学。

我们希望通过这些举措，能在较短的时间，为我国培养一大批高质量的信息技术人才，提高我国软件人才的国际竞争力，促进我国信息产业的快速发展，加快推动国家信息化进程，进而带动整个国民经济的跨越式发展。

教育部高等教育司

二〇〇二年三月

Preface

This book is intended as a textbook for a one- or two-semester course in compilers. Students will see the theory behind different components of a compiler, the programming techniques used to put the theory into practice, and the interfaces used to modularize the compiler. To make the interfaces and programming examples clear and concrete, we have written them in Java. Another edition of this book is available that uses the ML language.

Implementation project. The “student project compiler” that we have outlined is reasonably simple, but is organized to demonstrate some important techniques that are now in common use: abstract syntax trees to avoid tangling syntax and semantics, separation of instruction selection from register allocation, copy propagation to give flexibility to earlier phases of the compiler, and containment of target-machine dependencies. Unlike many “student compilers” found in other textbooks, this one has a simple but sophisticated back end, allowing good register allocation to be done after instruction selection.

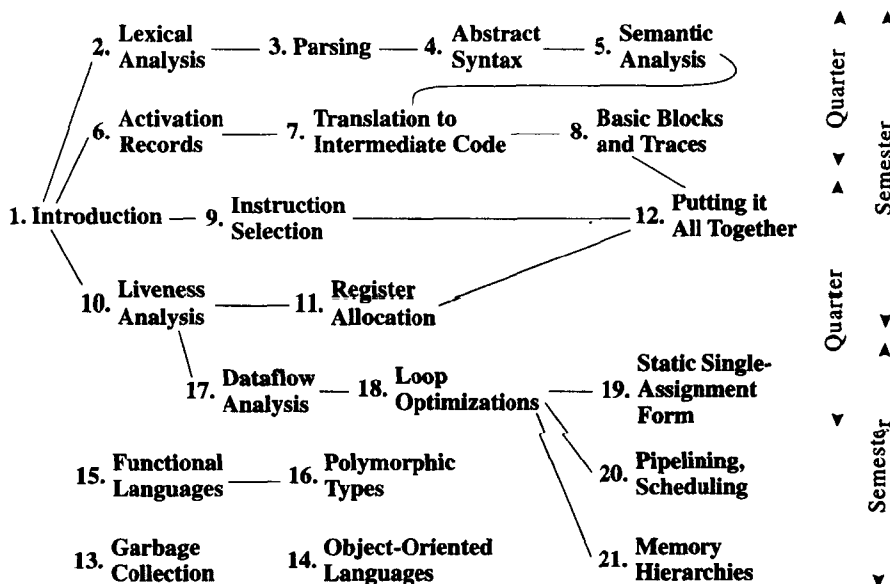
This second edition of the book has a redesigned project compiler: It uses a subset of Java, called MiniJava, as the source language for the compiler project, it explains the use of the parser generators JavaCC and SableCC, and it promotes programming with the Visitor pattern. Students using this edition can implement a compiler for a language they’re familiar with, using standard tools, in a more object-oriented style.

Each chapter in Part I has a programming exercise corresponding to one module of a compiler. Software useful for the exercises can be found at <http://uk.cambridge.org/resources/052182060X> (outside North America); <http://us.cambridge.org/titles/052182060X.html> (within North America).

Exercises. Each chapter has pencil-and-paper exercises; those marked with a star are more challenging, two-star problems are difficult but solvable, and

the occasional three-star exercises are not known to have a solution.

Course sequence. The figure shows how the chapters depend on each other.



- A one-semester course could cover all of Part I (Chapters 1–12), with students implementing the project compiler (perhaps working in groups); in addition, lectures could cover selected topics from Part II.
- An advanced or graduate course could cover Part II, as well as additional topics from the current literature. Many of the Part II chapters can stand independently from Part I, so that an advanced course could be taught to students who have used a different book for their first course.
- In a two-quarter sequence, the first quarter could cover Chapters 1–8, and the second quarter could cover Chapters 9–12 and some chapters from Part II.

Acknowledgments. Many people have provided constructive criticism or helped us in other ways on this book. Vidyut Samanta helped tremendously with both the text and the software for the new edition of the book. We would also like to thank Leonor Abraido-Fandino, Scott Ananian, Nils Andersen, Stephen Bailey, Joao Cangussu, Maia Ginsburg, Max Hailperin, David Hanson, Jeffrey Hsu, David MacQueen, Torben Mogensen, Doug Morgan, Robert Netzer, Elma Lee Noah, Mikael Petterson, Benjamin Pierce, Todd Proebsting, Anne Rogers, Barbara Ryder, Amr Sabry, Mooly Sagiv, Zhong Shao, Mary Lou Soffa, Andrew Tolmach, Kwangkeun Yi, and Kenneth Zadeck.

Contents

Preface

ix

Part I Fundamentals of Compilation

1	Introduction	3
1.1	Modules and interfaces	4
1.2	Tools and software	5
1.3	Data structures for tree languages	7
2	Lexical Analysis	16
2.1	Lexical tokens	17
2.2	Regular expressions	18
2.3	Finite automata	21
2.4	Nondeterministic finite automata	24
2.5	Lexical-analyzer generators	30
3	Parsing	38
3.1	Context-free grammars	40
3.2	Predictive parsing	45
3.3	LR parsing	55
3.4	Using parser generators	68
3.5	Error recovery	76
4	Abstract Syntax	86
4.1	Semantic actions	86
4.2	Abstract parse trees	89
4.3	Visitors	93
5	Semantic Analysis	103
5.1	Symbol tables	103

5.2	Type-checking MiniJava	111
6	Activation Records	116
6.1	Stack frames	118
6.2	Frames in the MiniJava compiler	126
7	Translation to Intermediate Code	136
7.1	Intermediate representation trees	137
7.2	Translation into trees	140
7.3	Declarations	155
8	Basic Blocks and Traces	162
8.1	Canonical trees	163
8.2	Taming conditional branches	169
9	Instruction Selection	176
9.1	Algorithms for instruction selection	179
9.2	CISC machines	187
9.3	Instruction selection for the MiniJava compiler	190
10	Liveness Analysis	203
10.1	Solution of dataflow equations	205
10.2	Liveness in the MiniJava compiler	214
11	Register Allocation	219
11.1	Coloring by simplification	220
11.2	Coalescing	223
11.3	Precolored nodes	227
11.4	Graph-coloring implementation	232
11.5	Register allocation for trees	241
12	Putting It All Together	249
 Part II Advanced Topics		
13	Garbage Collection	257
13.1	Mark-and-sweep collection	257
13.2	Reference counts	262
13.3	Copying collection	264

13.4	Generational collection	269
13.5	Incremental collection	272
13.6	Baker's algorithm	274
13.7	Interface to the compiler	275
14	Object-Oriented Languages	283
14.1	Class extension	283
14.2	Single inheritance of data fields	284
14.3	Multiple inheritance	286
14.4	Testing class membership	289
14.5	Private fields and methods	292
14.6	Classless languages	293
14.7	Optimizing object-oriented programs	293
15	Functional Programming Languages	298
15.1	A simple functional language	299
15.2	Closures	301
15.3	Immutable variables	302
15.4	Inline expansion	308
15.5	Closure conversion	316
15.6	Efficient tail recursion	319
15.7	Lazy evaluation	321
16	Polymorphic Types	335
16.1	Parametric polymorphism	336
16.2	Polymorphic type-checking	339
16.3	Translation of polymorphic programs	344
16.4	Resolution of static overloading	347
17	Dataflow Analysis	350
17.1	Intermediate representation for flow analysis	351
17.2	Various dataflow analyses	354
17.3	Transformations using dataflow analysis	359
17.4	Speeding up dataflow analysis	360
17.5	Alias analysis	369
18	Loop Optimizations	376
18.1	Dominators	379
18.2	Loop-invariant computations	384

18.3	Induction variables	385
18.4	Array-bounds checks	391
18.5	Loop unrolling	395
19	Static Single-Assignment Form	399
19.1	Converting to SSA form	402
19.2	Efficient computation of the dominator tree	410
19.3	Optimization algorithms using SSA	417
19.4	Arrays, pointers, and memory	423
19.5	The control-dependence graph	425
19.6	Converting back from SSA form	428
19.7	A functional intermediate form	430
20	Pipelining and Scheduling	440
20.1	Loop scheduling without resource bounds	444
20.2	Resource-bounded loop pipelining	448
20.3	Branch prediction	456
21	The Memory Hierarchy	464
21.1	Cache organization	465
21.2	Cache-block alignment	468
21.3	Prefetching	470
21.4	Loop interchange	476
21.5	Blocking	477
21.6	Garbage collection and the memory hierarchy	480
	Appendix: MiniJava Language Reference Manual	484
A.1	Lexical Issues	484
A.2	Grammar	484
A.3	Sample Program	486
	<i>Bibliography</i>	487
	<i>Index</i>	495

PART ONE

Fundamentals of Compilation

Introduction

A **compiler** was originally a program that “compiled” subroutines [a link-loader]. When in 1954 the combination “algebraic compiler” came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

Bauer and Eickel [1975]

This book describes techniques, data structures, and algorithms for translating programming languages into executable code. A modern compiler is often organized into many phases, each operating on a different abstract “language.” The chapters of this book follow the organization of a compiler, each covering a successive phase.

To illustrate the issues in compiling real programming languages, we show how to compile MiniJava, a simple but nontrivial subset of Java. Programming exercises in each chapter call for the implementation of the corresponding phase; a student who implements all the phases described in Part I of the book will have a working compiler. MiniJava is easily extended to support class extension or higher-order functions, and exercises in Part II show how to do this. Other chapters in Part II cover advanced techniques in program optimization. Appendix A describes the MiniJava language.

The interfaces between modules of the compiler are almost as important as the algorithms inside the modules. To describe the interfaces concretely, it is useful to write them down in a real programming language. This book uses Java – a simple object-oriented language. Java is *safe*, in that programs cannot circumvent the type system to violate abstractions; and it has garbage collection, which greatly simplifies the management of dynamic storage al-

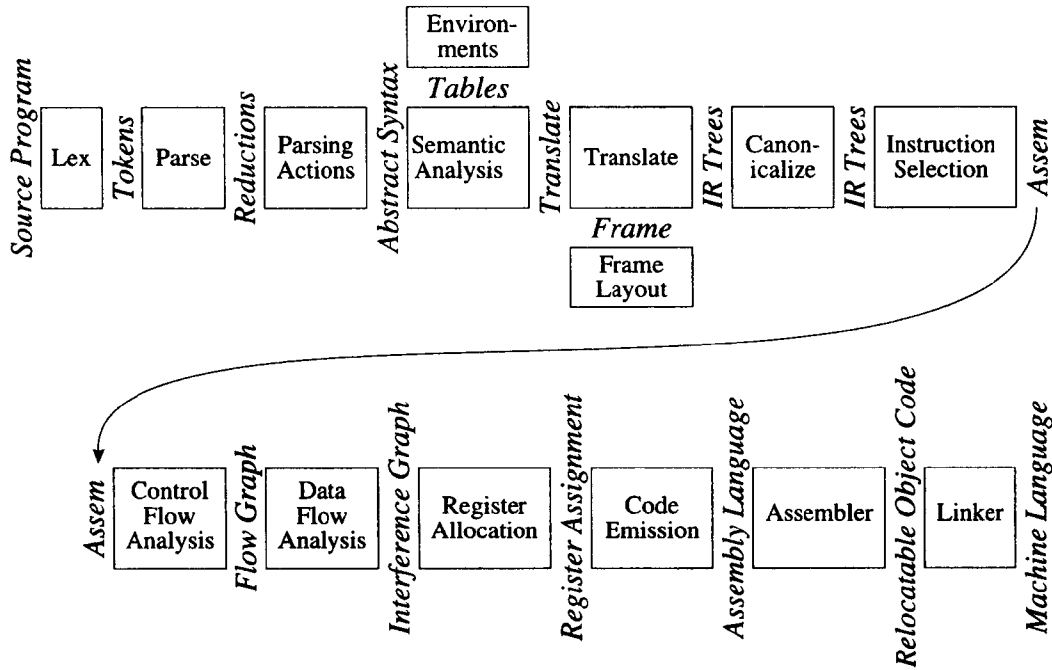


FIGURE 1.1. Phases of a compiler, and interfaces between them.

location. Both of these properties are useful in writing compilers (and almost any kind of software).

This is not a textbook on Java programming. Students using this book who do not know Java already should pick it up as they go along, using a Java programming book as a reference. Java is a small enough language, with simple enough concepts, that this should not be difficult for students with good programming skills in other languages.

1.1

MODULES AND INTERFACES

Any large software system is much easier to understand and implement if the designer takes care with the fundamental abstractions and interfaces. Figure 1.1 shows the phases in a typical compiler. Each phase is implemented as one or more software modules.

Breaking the compiler into this many pieces allows for reuse of the components. For example, to change the target machine for which the compiler pro-

duces machine language, it suffices to replace just the Frame Layout and Instruction Selection modules. To change the source language being compiled, only the modules up through Translate need to be changed. The compiler can be attached to a language-oriented syntax editor at the *Abstract Syntax* interface.

The learning experience of coming to the right abstraction by several iterations of *think–implement–redesign* is one that should not be missed. However, the student trying to finish a compiler project in one semester does not have this luxury. Therefore, we present in this book the outline of a project where the abstractions and interfaces are carefully thought out, and are as elegant and general as we are able to make them.

Some of the interfaces, such as *Abstract Syntax*, *IR Trees*, and *Assem*, take the form of data structures: For example, the Parsing Actions phase builds an *Abstract Syntax* data structure and passes it to the Semantic Analysis phase. Other interfaces are abstract data types; the *Translate* interface is a set of functions that the Semantic Analysis phase can call, and the *Tokens* interface takes the form of a function that the Parser calls to get the next token of the input program.

DESCRIPTION OF THE PHASES

Each chapter of Part I of this book describes one compiler phase, as shown in Table 1.2

This modularization is typical of many real compilers. But some compilers combine Parse, Semantic Analysis, Translate, and Canonicalize into one phase; others put Instruction Selection much later than we have done, and combine it with Code Emission. Simple compilers omit the Control Flow Analysis, Data Flow Analysis, and Register Allocation phases.

We have designed the compiler in this book to be as simple as possible, but no simpler. In particular, in those places where corners are cut to simplify the implementation, the structure of the compiler allows for the addition of more optimization or fancier semantics without violence to the existing interfaces.

Two of the most useful abstractions used in modern compilers are *context-free grammars*, for parsing, and *regular expressions*, for lexical analysis. To make the best use of these abstractions it is helpful to have special tools,

Chapter	Phase	Description
2	Lex	Break the source file into individual words, or <i>tokens</i> .
3	Parse	Analyze the phrase structure of the program.
4	Semantic Actions	Build a piece of <i>abstract syntax tree</i> corresponding to each phrase.
5	Semantic Analysis	Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase.
6	Frame Layout	Place variables, function-parameters, etc. into activation records (stack frames) in a machine-dependent way.
7	Translate	Produce <i>intermediate representation trees</i> (IR trees), a notation that is not tied to any particular source language or target-machine architecture.
8	Canonicalize	Hoist side effects out of expressions, and clean up conditional branches, for the convenience of the next phases.
9	Instruction Selection	Group the IR-tree nodes into clumps that correspond to the actions of target-machine instructions.
10	Control Flow Analysis	Analyze the sequence of instructions into a <i>control flow graph</i> that shows all the possible flows of control the program might follow when it executes.
10	Dataflow Analysis	Gather information about the flow of information through variables of the program; for example, <i>liveness analysis</i> calculates the places where each program variable holds a still-needed value (is <i>live</i>).
11	Register Allocation	Choose a register to hold each of the variables and temporary values used by the program; variables not live at the same time can share the same register.
12	Code Emission	Replace the temporary names in each machine instruction with machine registers.

TABLE 1.2. Description of compiler phases.

such as *Yacc* (which converts a grammar into a parsing program) and *Lex* (which converts a declarative specification into a lexical-analysis program). Fortunately, such tools are available for Java, and the project described in this book makes use of them.

The programming projects in this book can be compiled using any Java