# modern compiler implementation in Java

second edition

andrew w. appel

CAMBRIDGE

more information - www.cambridge.org/052182060X

This page intentionally left blank

# Modern Compiler Implementation in Java
# Second Edition

This textbook describes all phases of a compiler: lexical analysis, parsing, abstract syntax, semantic actions, intermediate representations, instruction selection via tree matching, dataflow analysis, graph-coloring register allocation, and runtime systems. It includes good coverage of current techniques in code generation and register allocation, as well as the compilation of functional and object-oriented languages, which is missing from most books. The most accepted and successful techniques are described concisely, rather than as an exhaustive catalog of every possible variant. Detailed descriptions of the interfaces between modules of a compiler are illustrated with actual Java classes.

The first part of the book, Fundamentals of Compilation, is suitable for a one-semester first course in compiler design. The second part, Advanced Topics, which includes the compilation of object-oriented and functional languages, garbage collection, loop optimization, SSA form, instruction scheduling, and optimization for cache-memory hierarchies, can be used for a second-semester or graduate course.

This new edition has been rewritten extensively to include more discussion of Java and object-oriented programming concepts, such as visitor patterns. A unique feature is the newly redesigned compiler project in Java for a subset of Java itself. The project includes both front-end and back-end phases, so that students can build a complete working compiler in one semester.

Andrew W. Appel is Professor of Computer Science at Princeton University. He has done research and published papers on compilers, functional programming languages, runtime systems and garbage collection, type systems, and computer security; he is also author of the book *Compiling with Continuations.* He is a designer and founder of the Standard ML of New Jersey project. In 1998, Appel was elected a Fellow of the Association for Computing Machinery for "significant research contributions in the area of programming languages and compilers" and for his work as editor-in-chief (1993–97) of the *ACM Transactions on Programming Languages and Systems,* the leading journal in the field of compilers and programming languages.

Jens Palsberg is Associate Professor of Computer Science at Purdue University. His research interests are programming languages, compilers, software engineering, and information security. He has authored more than 50 technical papers in these areas and a book with Michael Schwartzbach, *Object-oriented Type Systems.* In 1998, he received the National Science Foundation Faculty Early Career Development Award, and in 1999, the Purdue University Faculty Scholar award.

# Modern Compiler Implementation in Java

## Second Edition

ANDREW W. APPEL
*Princeton University*

with JENS PALSBERG
*Purdue University*

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, UK
40 West 20th Street, New York, NY 10011-4211, USA
477 Williamstown Road, Port Melbourne, VIC 3207, Australia
Ruiz de Alarcón 13, 28014 Madrid, Spain
Dock House, The Waterfront, Cape Town 8001, South Africa

http://www.cambridge.org



First published in printed format 2002

ISBN 0-511-04286-8 eBook (netLibrary)
ISBN 0-521-82060-X hardback

# 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

*Bibliography* 487

*Index* 495

# 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.

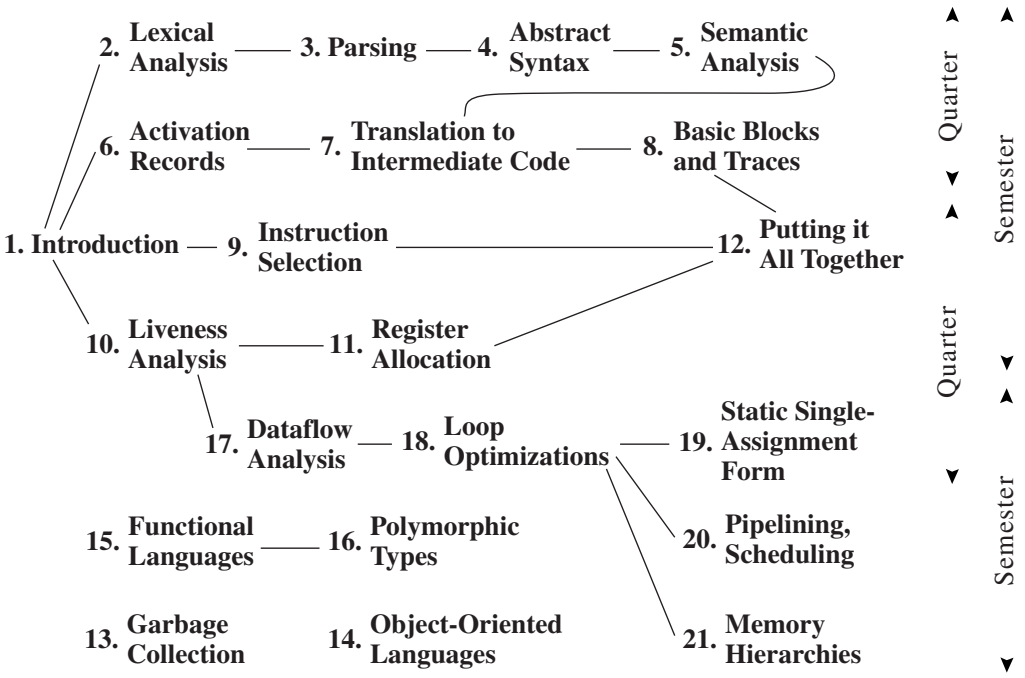
This second edition of the book has a redesigned project compiler: It uses a subset of Java, called MiniJava, as the source language for the compiler project, it explains the use of the parser generators JavaCC and SableCC, and it promotes programming with the Visitor pattern. Students using this edition 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.

# PART ONE

# Fundamentals of Compilation

# 1 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-

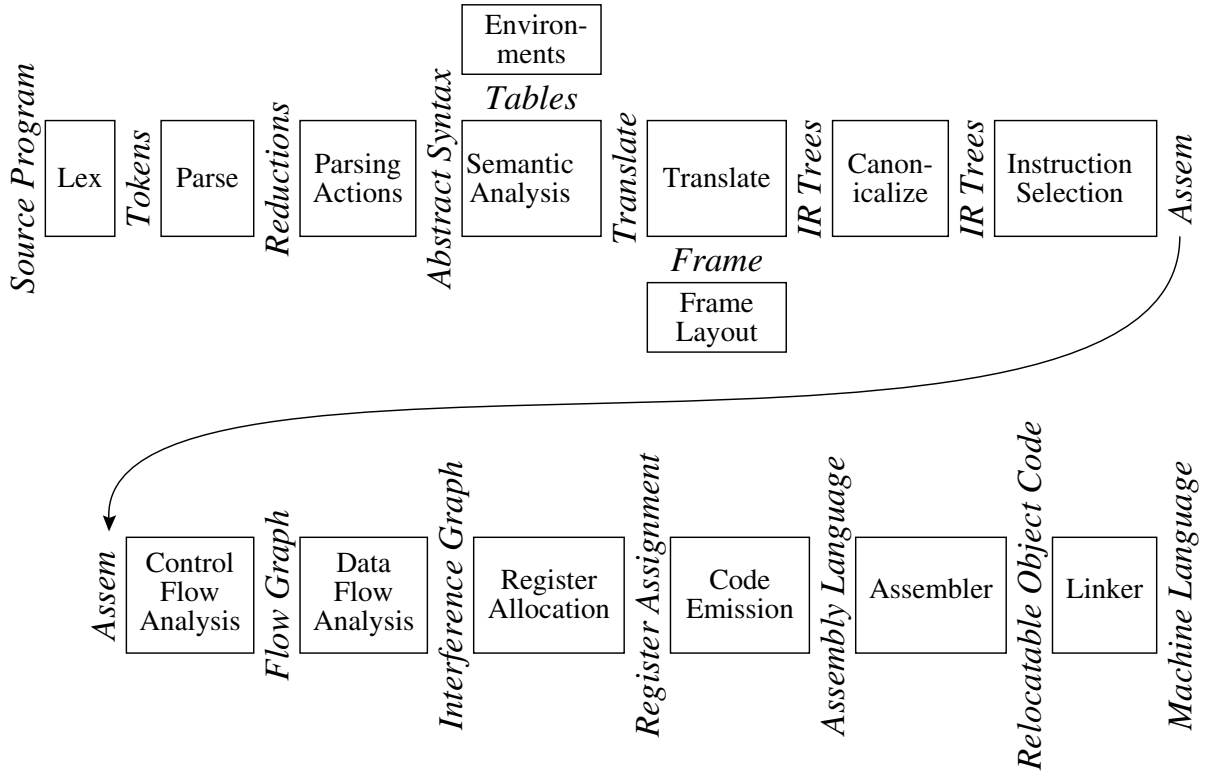



**FIGURE 1.1.** Phases of a compiler, and interfaces between them.

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.

## 1.2 TOOLS AND SOFTWARE

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 *tokens*. |
| 3 | Parse | Analyze the phrase structure of the program. |
| 4 | Semantic Actions | Build a piece of *abstract syntax tree* 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 *intermediate representation trees* (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 *control flow graph* 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, *liveness analysis* calculates the places where each program variable holds a still-needed value (is *live*). |
| 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

| | | | |
|---|---|---|---|
| *Stm* → *Stm* ; *Stm* | (CompoundStm) | *ExpList* → *Exp* , *ExpList* | (PairExpList) |
| *Stm* → `id :=` *Exp* | (AssignStm) | *ExpList* → *Exp* | (LastExpList) |
| *Stm* → `print (` *ExpList* `)` | (PrintStm) | *Binop* → + | (Plus) |
| *Exp* → `id` | (IdExp) | *Binop* → − | (Minus) |
| *Exp* → `num` | (NumExp) | *Binop* → × | (Times) |
| *Exp* → *Exp Binop Exp* | (OpExp) | *Binop* → / | (Div) |
| *Exp* → ( *Stm* , *Exp* ) | (EseqExp) | | |

**GRAMMAR 1.3.** A straight-line programming language.

compiler. The parser generators *JavaCC* and *SableCC* are freely available on the Internet; for information see the World Wide Web page

http://uk.cambridge.org/resources/052182060X (outside North America);
http://us.cambridge.org/titles/052182060X.html (within North America).

Source code for some modules of the MiniJava compiler, skeleton source code and support code for some of the programming exercises, example MiniJava programs, and other useful files are also available from the same Web address. The programming exercises in this book refer to this directory as `$MINIJAVA/` when referring to specific subdirectories and files contained therein.

## 1.3 DATA STRUCTURES FOR TREE LANGUAGES

Many of the important data structures used in a compiler are *intermediate representations* of the program being compiled. Often these representations take the form of trees, with several node types, each of which has different attributes. Such trees can occur at many of the phase-interfaces shown in Figure 1.1.

Tree representations can be described with grammars, just like programming languages. To introduce the concepts, we will show a simple programming language with statements and expressions, but no loops or if-statements (this is called a language of *straight-line programs*).

The syntax for this language is given in Grammar 1.3.

The informal semantics of the language is as follows. Each *Stm* is a statement, each *Exp* is an expression. $s_1; s_2$ executes statement $s_1$, then statement $s_2$. $i := e$ evaluates the expression $e$, then "stores" the result in variable $i$.

`print`$(e_1, e_2, \ldots, e_n)$ displays the values of all the expressions, evaluated left to right, separated by spaces, terminated by a newline.

An *identifier expression*, such as $i$, yields the current contents of the variable $i$. A *number* evaluates to the named integer. An *operator expression* $e_1$ op $e_2$ evaluates $e_1$, then $e_2$, then applies the given binary operator. And an *expression sequence* $(s, e)$ behaves like the C-language "comma" operator, evaluating the statement $s$ for side effects before evaluating (and returning the result of) the expression $e$.

For example, executing this program

```
a := 5+3; b := (print(a, a-1), 10*a); print(b)
```

prints

```
8 7
80
```

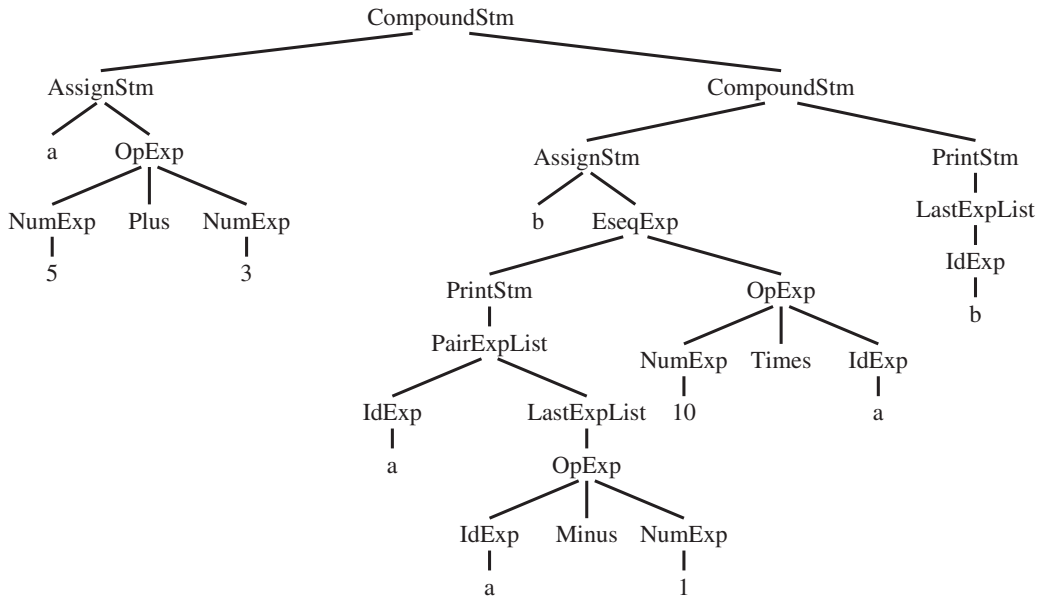
How should this program be represented inside a compiler? One representation is *source code*, the characters that the programmer writes. But that is not so easy to manipulate. More convenient is a tree data structure, with one node for each statement (Stm) and expression (Exp). Figure 1.4 shows a tree representation of the program; the nodes are labeled by the production labels of Grammar 1.3, and each node has as many children as the corresponding grammar production has right-hand-side symbols.

We can translate the grammar directly into data structure definitions, as shown in Program 1.5. Each grammar symbol corresponds to an `abstract class` in the data structures:

| Grammar | class |
|---|---|
| *Stm* | Stm |
| *Exp* | Exp |
| *ExpList* | ExpList |
| *id* | String |
| *num* | int |

For each grammar rule, there is one *constructor* that belongs to the class for its left-hand-side symbol. We simply *extend* the abstract class with a "concrete" class for each grammar rule. The constructor (class) names are indicated on the right-hand side of Grammar 1.3.

Each grammar rule has right-hand-side components that must be represented in the data structures. The CompoundStm has two Stm's on the right-hand side; the AssignStm has an identifier and an expression; and so on.




```
a := 5 + 3 ; b := ( print ( a , a - 1 ) , 10 * a ) ; print ( b )
```

**FIGURE 1.4.** Tree representation of a straight-line program.

These become *fields* of the subclasses in the Java data structure. Thus, CompoundStm has two fields (also called *instance variables*) called `stm1` and `stm2`; AssignStm has fields `id` and `exp`.

For Binop we do something simpler. Although we could make a Binop class – with subclasses for Plus, Minus, Times, Div – this is overkill because none of the subclasses would need any fields. Instead we make an "enumeration" type (in Java, actually an integer) of constants (`final int` variables) local to the OpExp class.

**Programming style.** We will follow several conventions for representing tree data structures in Java:

1. Trees are described by a grammar.
2. A tree is described by one or more abstract classes, each corresponding to a symbol in the grammar.
3. Each abstract class is *extended* by one or more subclasses, one for each grammar rule.

```
public abstract class Stm {}

public class CompoundStm extends Stm {
   public Stm stm1, stm2;
   public CompoundStm(Stm s1, Stm s2) {stm1=s1; stm2=s2;}}

public class AssignStm extends Stm {
   public String id; public Exp exp;
   public AssignStm(String i, Exp e) {id=i; exp=e;}}

public class PrintStm extends Stm {
   public ExpList exps;
   public PrintStm(ExpList e) {exps=e;}}

public abstract class Exp {}

public class IdExp extends Exp {
   public String id;
   public IdExp(String i) {id=i;}}

public class NumExp extends Exp {
   public int num;
   public NumExp(int n) {num=n;}}

public class OpExp extends Exp {
   public Exp left, right; public int oper;
   final public static int Plus=1,Minus=2,Times=3,Div=4;
   public OpExp(Exp l, int o, Exp r) {left=l; oper=o; right=r;}}

public class EseqExp extends Exp {
   public Stm stm; public Exp exp;
   public EseqExp(Stm s, Exp e) {stm=s; exp=e;}}

public abstract class ExpList {}

public class PairExpList extends ExpList {
   public Exp head; public ExpList tail;
   public PairExpList(Exp h, ExpList t) {head=h; tail=t;}}

public class LastExpList extends ExpList {
   public Exp head;
   public LastExpList(Exp h) {head=h;}}
```

**PROGRAM 1.5.** Representation of straight-line programs.

4. For each nontrivial symbol in the right-hand side of a rule, there will be one field in the corresponding class. (A trivial symbol is a punctuation symbol such as the semicolon in CompoundStm.)
5. Every class will have a constructor function that initializes all the fields.
6. Data structures are initialized when they are created (by the constructor functions), and are never modified after that (until they are eventually discarded).

**Modularity principles for Java programs.** A compiler can be a big program; careful attention to modules and interfaces prevents chaos. We will use these principles in writing a compiler in Java:

1. Each phase or module of the compiler belongs in its own package.
2. "Import on demand" declarations will not be used. If a Java file begins with
```
import A.F.*; import A.G.*; import B.*; import C.*;
```
then the human reader *will have to look outside this file* to tell which package defines the `X` that is used in the expression `X.put()`.
3. "Single-type import" declarations are a better solution. If the module begins
```
import A.F.W; import A.G.X; import B.Y; import C.Z;
```
then you can tell *without looking outside this file* that `X` comes from `A.G`.
4. Java is naturally a multithreaded system. We would like to support multiple simultaneous compiler threads and compile two different programs simultaneously, one in each compiler thread. Therefore, static variables must be avoided unless they are `final` (constant). We never want two compiler threads to be updating the same (static) instance of a variable.

## PROGRAM STRAIGHT-LINE PROGRAM INTERPRETER

Implement a simple program analyzer and interpreter for the straight-line programming language. This exercise serves as an introduction to *environments* (symbol tables mapping variable names to information about the variables); to *abstract syntax* (data structures representing the phrase structure of programs); to *recursion over tree data structures*, useful in many parts of a compiler; and to a *functional style* of programming without assignment statements.

It also serves as a "warm-up" exercise in Java programming. Programmers experienced in other languages but new to Java should be able to do this exercise, but will need supplementary material (such as textbooks) on Java.

Programs to be interpreted are already parsed into abstract syntax, as described by the data types in Program 1.5.

However, we do not wish to worry about parsing the language, so we write this program by applying data constructors:

```
Stm prog =
new CompoundStm(new AssignStm("a",
                      new OpExp(new NumExp(5),
                                OpExp.Plus, new NumExp(3))),
 new CompoundStm(new AssignStm("b",
    new EseqExp(new PrintStm(new PairExpList(new IdExp("a"),
              new LastExpList(new OpExp(new IdExp("a"),
                              OpExp.Minus,new NumExp(1))))),

          new OpExp(new NumExp(10), OpExp.Times,
                    new IdExp("a")))),
  new PrintStm(new LastExpList(new IdExp("b")))));
```

Files with the data type declarations for the trees, and this sample program, are available in the directory `$MINIJAVA/chap1`.

Writing interpreters without side effects (that is, assignment statements that update variables and data structures) is a good introduction to *denotational semantics* and *attribute grammars*, which are methods for describing what programming languages do. It's often a useful technique in writing compilers, too; compilers are also in the business of saying what programming languages do.

Therefore, in implementing these programs, never assign a new value to any variable or object field except when it is initialized. For local variables, use the initializing form of declaration (for example, `int i=j+3;`) and for each class, make a constructor function (like the `CompoundStm` constructor in Program 1.5).

1. Write a Java function `int maxargs(Stm s)` that tells the maximum number of arguments of any `print` statement within any subexpression of a given statement. For example, `maxargs(prog)` is 2.
2. Write a Java function `void interp(Stm s)` that "interprets" a program in this language. To write in a "functional programming" style – in which you never use an assignment statement – initialize each local variable as you declare it.

Your functions that examine each `Exp` will have to use `instanceof` to determine which subclass the expression belongs to and then cast to the proper subclass. Or you can add methods to the `Exp` and `Stm` classes to avoid the use of `instanceof`.

For part 1, remember that print statements can contain expressions that contain other print statements.

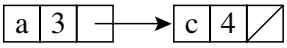
For part 2, make two mutually recursive functions `interpStm` and `interpExp`. Represent a "table," mapping identifiers to the integer values assigned to them, as a list of `id` × `int` pairs.

```
class Table {
  String id; int value; Table tail;
  Table(String i, int v, Table t) {id=i; value=v; tail=t;}
}
```

Then `interpStm` is declared as

```
Table interpStm(Stm s, Table t)
```

taking a table $t_1$ as argument and producing the new table $t_2$ that's just like $t_1$ except that some identifiers map to different integers as a result of the statement.

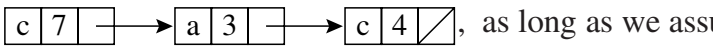
For example, the table $t_1$ that maps $a$ to 3 and maps $c$ to 4, which we write $\{a \mapsto 3, c \mapsto 4\}$ in mathematical notation, could be represented as the linked list 

.

Now, let the table $t_2$ be just like $t_1$, except that it maps $c$ to 7 instead of 4. Mathematically, we could write,

$$t_2 = \text{update}(t_1, c, 7),$$

where the update function returns a new table $\{a \mapsto 3, c \mapsto 7\}$.

On the computer, we could implement $t_2$ by putting a new cell at the head of the linked list: 

, as long as we assume that the *first* occurrence of $c$ in the list takes precedence over any later occurrence.

Therefore, the `update` function is easy to implement; and the corresponding `lookup` function

```
int lookup(Table t, String key)
```

just searches down the linked list. Of course, in an object-oriented style, `int lookup(String key)` should be a method of the `Table` class.

Interpreting expressions is more complicated than interpreting statements, because expressions return integer values *and* have side effects. We wish to simulate the straight-line programming language's assignment statements without doing any side effects in the interpreter itself. (The `print` statements will be accomplished by interpreter side effects, however.) The solution is to declare `interpExp` as

```
class IntAndTable {int i; Table t;
    IntAndTable(int ii, Table tt) {i=ii; t=tt;}
   }
IntAndTable interpExp(Exp e, Table t) ···
```

The result of interpreting an expression $e_1$ with table $t_1$ is an integer value $i$ and a new table $t_2$. When interpreting an expression with two subexpressions (such as an `OpExp`), the table $t_2$ resulting from the first subexpression can be used in processing the second subexpression.

## EXERCISES

**1.1** This simple program implements *persistent* functional binary search trees, so that if `tree2=insert(x,tree1)`, then `tree1` is still available for lookups even while `tree2` can be used.

```
class Tree {Tree left; String key; Tree right;
   Tree(Tree l, String k, Tree r) {left=l; key=k; right=r;}

Tree insert(String key, Tree t) {
  if (t==null) return new Tree(null, key, null)
  else if (key.compareTo(t.key) < 0)
        return new Tree(insert(key,t.left),t.key,t.right);
  else if (key.compareTo(t.key) > 0)
        return new Tree(t.left,t.key,insert(key,t.right));
  else return new Tree(t.left,key,t.right);
}
```

a. Implement a `member` function that returns `true` if the item is found, else `false`.

b. Extend the program to include not just membership, but the mapping of keys to bindings:

```
Tree insert(String key, Object binding, Tree t);
Object lookup(String key, Tree t);
```

c. These trees are not balanced; demonstrate the behavior on the following two sequences of insertions:
(a) t s p i p f b s t
(b) a b c d e f g h i

*d. Research balanced search trees in Sedgewick [1997] and recommend a balanced-tree data structure for functional symbol tables. **Hint:** To preserve a functional style, the algorithm should be one that rebalances

on insertion but not on lookup, so a data structure such as *splay trees* is not appropriate.

e. Rewrite in an object-oriented (but still "functional") style, so that insertion is now `t.insert(key)` instead of `insert(key,t)`. **Hint:** You'll need an `EmptyTree` subclass.