Modern Compiler Implementation in Java

Basic Techniques

ANDREW W. APPEL
Princeton University

Preliminary edition of *Modern Compiler Implementation in Java*

CAMBRIDGE
UNIVERSITY PRESS

PUBLISHED BY THE PRESS SYNDICATE OF THE UNIVERSITY OF CAMBRIDGE
The Pitt Building, Trumpington Street, Cambridge CB2 1RP, United Kingdom

CAMBRIDGE UNIVERSITY PRESS
The Edinburgh Building, Cambridge CB2 2RU, United Kingdom
40 West 20th Street, New York, NY 10011-4211, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

© Andrew W. Appel, 1997

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.

First published 1997

Printed in the United States of America

Typeset in Times, Courier, and Optima

Library of Congress Cataloguing-in-Publication data applied for

A catalog record for this book is available from the British Library

0-521-58275-X	Modern Compiler Implementation in ML: Basic Techniques (hardback)
0-521-58775-1	Modern Compiler Implementation in ML: Basic Techniques (paperback)
0-521-58387-X	Modern Compiler Implementation in Java: Basic Techniques (hardback)
0-521-58654-2	Modern Compiler Implementation in Java: Basic Techniques (paperback)
0-521-58389-6	Modern Compiler Implementation in C: Basic Techniques (hardback)
0-521-58653-4	Modern Compiler Implementation in C: Basic Techniques (paperback)

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 JavaLex: a lexical analyzer generator 31

3 Parsing 40
- 3.1 Context-free grammars 42
- 3.2 Predictive parsing 47
- 3.3 LR parsing 57
- 3.4 Using parser generators 68

4 Abstract Syntax 80
- 4.1 Semantic actions 80
- 4.2 Abstract parse trees 87

5 Semantic Analysis 99
- 5.1 Symbol tables 99
- 5.2 Bindings for the Tiger compiler 107
- 5.3 Type-checking expressions 110

5.4 Type-checking declarations 114

6 Activation Records 120
6.1 Stack frames 122
6.2 Frames in the Tiger compiler 130

7 Translation to Intermediate Code 144
7.1 Intermediate representation trees 145
7.2 Translation into trees 148
7.3 Declarations 164

8 Basic Blocks and Traces 169
8.1 Canonical trees 170
8.2 Taming conditional branches 176

9 Instruction Selection 181
9.1 Algorithms for instruction selection 184
9.2 CISC machines 192
9.3 Instruction selection for the Tiger compiler 195

10 Liveness Analysis 207
10.1 Solution of dataflow equations 209
10.2 Liveness in the Tiger compiler 217

11 Register Allocation 223
11.1 Coloring by simplification 224
11.2 Coalescing 227
11.3 Graph coloring implementation 232
11.4 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 271
13.6 Baker's algorithm 274
13.7 Interface to the compiler 275

14 Object-oriented Languages 283
14.1 Classes 283
14.2 Single inheritance of data fields 286
14.3 Multiple inheritance 288
14.4 Testing class membership 290
14.5 Private fields and methods 293
14.6 Classless languages 294
14.7 Optimizing object-oriented programs 295

15 Functional Programming Languages 299
15.1 A simple functional language 300
15.2 Closures 302
15.3 Immutable variables 303
15.4 Inline expansion 309
15.5 Closure conversion 315
15.6 Efficient tail recursion 318
15.7 Lazy evaluation 320

16 Dataflow Analysis 333
16.1 Intermediate representation for flow analysis 334
16.2 Various dataflow analyses 337
16.3 Transformations using dataflow analysis 341
16.4 Speeding up dataflow analysis 343
16.5 Alias analysis 351

17 Loop Optimizations 359
17.1 Dominators 362
17.2 Loop-invariant computations 365
17.3 Induction variables 369
17.4 Array bounds checks 374
17.5 Loop unrolling 378

Appendix: Tiger Language Reference Manual 381
A.1 Lexical issues 381
A.2 Declarations 381

A.3 Variables and expressions 384
A.4 Standard library 388

Bibliography 389

Index 393

Preface

Over the past decade, there have been several shifts in the way compilers are built. New kinds of programming languages are being used: object-oriented languages with dynamic methods, functional languages with nested scope and first-class function closures; and many of these languages require garbage collection. New machines have large register sets and a high penalty for memory access, and can often run much faster with compiler assistance in scheduling instructions and managing instructions and data for cache locality.

This book is intended as a textbook for a one-semester or two-quarter 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, I have written them in the Java programming language. Other editions of this book are available that use the C and ML languages.

The "student project compiler" that I 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, sophisticated copy propagation to allow greater flexibility to earlier phases of the compiler, and careful containment of target-machine dependencies to one module.

This book, *Modern Compiler Implementation in Java: Basic Techniques*, is the preliminary edition of a more complete book to be published in 1998, entitled *Modern Compiler Implementation in Java*. That book will have a more comprehensive set of exercises in each chapter, a "further reading" discussion at the end of every chapter, and another dozen chapters on advanced material not in this edition, such as parser error recovery, code-generator generators, byte-code interpreters, static single-assignment form, instruction scheduling and software pipelining, parallelization techniques, and cache-locality optimizations such as prefetching, blocking, instruction-cache layout, and branch prediction.

Exercises. Each of the chapters in Part I has a programming exercise corresponding to one module of a compiler. Unlike many "student project compilers" found in textbooks, this one has a simple but sophisticated back end, allowing good register allocation to be done after instruction selection. Software useful for the programming exercises can be found at

`http://www.cs.princeton.edu/~appel/modern/`

There are also pencil and paper exercises in each chapter; those marked with a star * are a bit more challenging, two-star problems are difficult but solvable, and the occasional three-star exercises are not known to have a solution.

Acknowledgments. Several people have provided constructive criticism, course-tested the manuscript, or helped in other ways in the production of this book. I would like to thank Stephen Bailey, Maia Ginsburg, David Hanson, Elma Lee Noah, Todd Proebsting, Barbara Ryder, Amr Sabry, Zhong Shao, Mary Lou Soffa, Andrew Tolmach, and Kwangkeun Yi.

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, I show how to compile Tiger, a simple but nontrivial language of the Algol family, with nested scope and heap-allocated records. 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. Tiger is easily modified to be *functional* or *object-oriented* (or both), 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 Tiger 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 allocation.

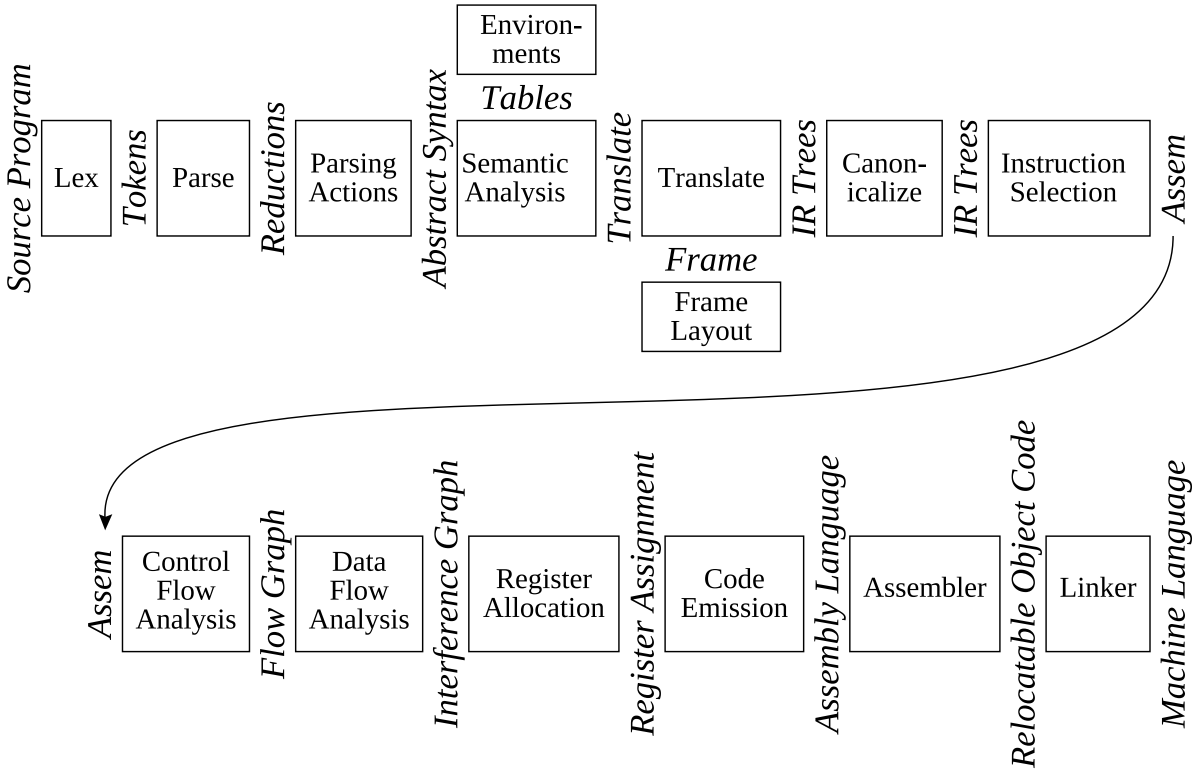

Source Program
Lex
Tokens
Parse
Reductions
Parsing Actions
Abstract Syntax
Environ-ments
Tables
Semantic Analysis
Translate
Translate
Frame
Frame Layout
IR Trees
Canon-icalize
IR Trees
Instruction Selection
Assem
Assem
Control Flow Analysis
Flow Graph
Data Flow Analysis
Interference Graph
Register Allocation
Register Assignment
Code Emission
Assembly Language
Assembler
Relocatable Object Code
Linker
Machine Language

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

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 produces 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, I 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 I am 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 I have done, and combine it with Code Emission. Simple compilers omit the Control Flow Analysis, Data Flow Analysis, and Register Allocation phases.

I 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 best use of these abstractions it is helpful to have special tools, such as *Yacc*

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.

(which converts a grammar into a parsing program) and *Lex* (which converts a declarative specification into a lexical analysis program). Fortunately, good versions of these 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 Sun's Java

$\mathit{Stm} \rightarrow \mathit{Stm}\ ;\ \mathit{Stm}$	(CompoundStm)
$\mathit{Stm} \rightarrow \texttt{id :=}\ \mathit{Exp}$	(AssignStm)
$\mathit{Stm} \rightarrow \texttt{print}\ (\ \mathit{ExpList}\ )$	(PrintStm)
$\mathit{Exp} \rightarrow \texttt{id}$	(IdExp)
$\mathit{Exp} \rightarrow \texttt{num}$	(NumExp)
$\mathit{Exp} \rightarrow \mathit{Exp}\ \mathit{Binop}\ \mathit{Exp}$	(OpExp)
$\mathit{Exp} \rightarrow (\ \mathit{Stm}\ ,\ \mathit{Exp}\ )$	(EseqExp)
$\mathit{ExpList} \rightarrow \mathit{Exp}\ ,\ \mathit{ExpList}$	(PairExpList)
$\mathit{ExpList} \rightarrow \mathit{Exp}$	(LastExpList)
$\mathit{Binop} \rightarrow +$	(Plus)
$\mathit{Binop} \rightarrow -$	(Minus)
$\mathit{Binop} \rightarrow \times$	(Times)
$\mathit{Binop} \rightarrow /$	(Div)

GRAMMAR 1.3. A straight-line programming language.

Development Kit, or (in principle) any Java compiler. The lexical-analyzer generator *JavaLex* and the parser generator *CUP* are freely available on the Internet; for information see the Wide-World Web page

`http://www.cs.princeton.edu/~appel/modern/`

Source code for some modules of the Tiger compiler, support code for some of the programming exercises, example Tiger programs, and other useful files are also available from the same Web address.

Skeleton source code for the programming assignments is available from this Web page; the programming exercises in this book refer to this directory as `$TIGER/` 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, I 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* $e_1 \text{ op } e_2$ 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

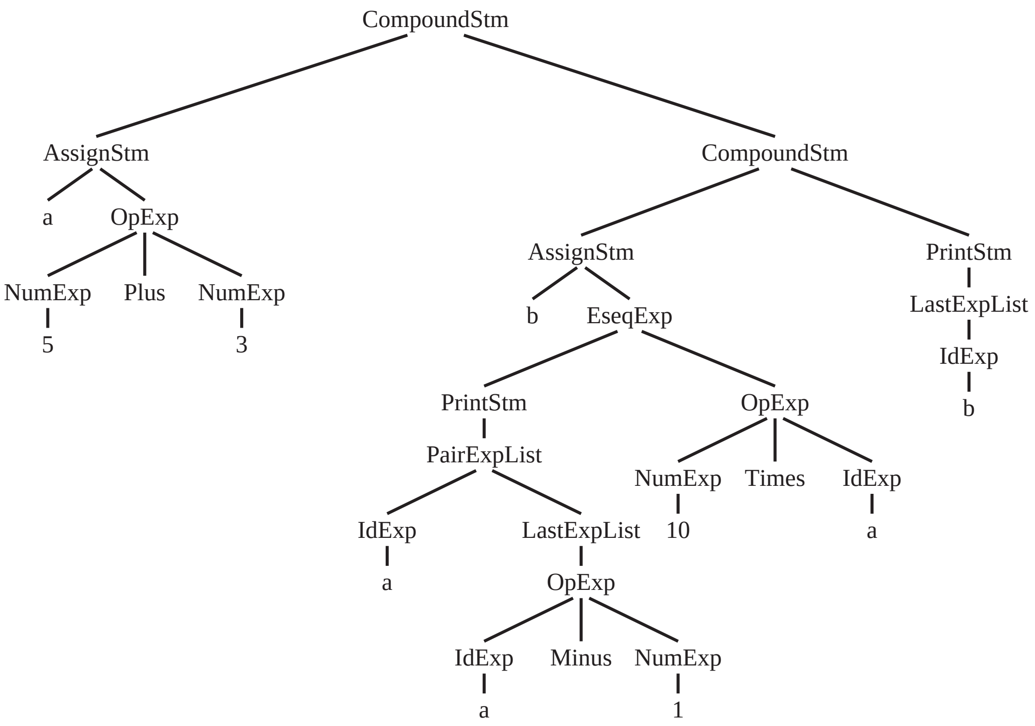
```
8 7
80
```

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 Figure 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.


CompoundStm
AssignStm
CompoundStm
a
OpExp
NumExp
Plus
NumExp
5
3
AssignStm
PrintStm
b
EseqExp
LastExpList
IdExp
b
PrintStm
OpExp
PairExpList
NumExp
Times
IdExp
10
a
IdExp
LastExpList
a
OpExp
IdExp
Minus
NumExp
a
1

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

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

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. 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.

```
public abstract class Stm {}

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

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

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

public abstract class Exp {}

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

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

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

public abstract class ExpList {}

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

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

PROGRAM 1.5. Representation of straight-line programs.

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.
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 `$TIGER/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.

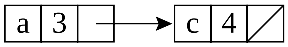
For part 2, make two mutually recursive functions `interpStm` and `interpExp`. Represent a "table," mapping identifiers to the integer values assigned to them, as a list of `id` $\times$ `int` pairs.

```
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)
```

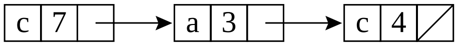
taking a table t_1 as argument and producing the new table t_2 that's just like t_1 except that some identifiers map to different integers as a result of the statement.

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,

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

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, `lookup` 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; 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 < t.key)
        return new Tree(insert(key,t.left),t.key,t.right);
  else if (key > t.key)
        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 [1988] 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.)

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