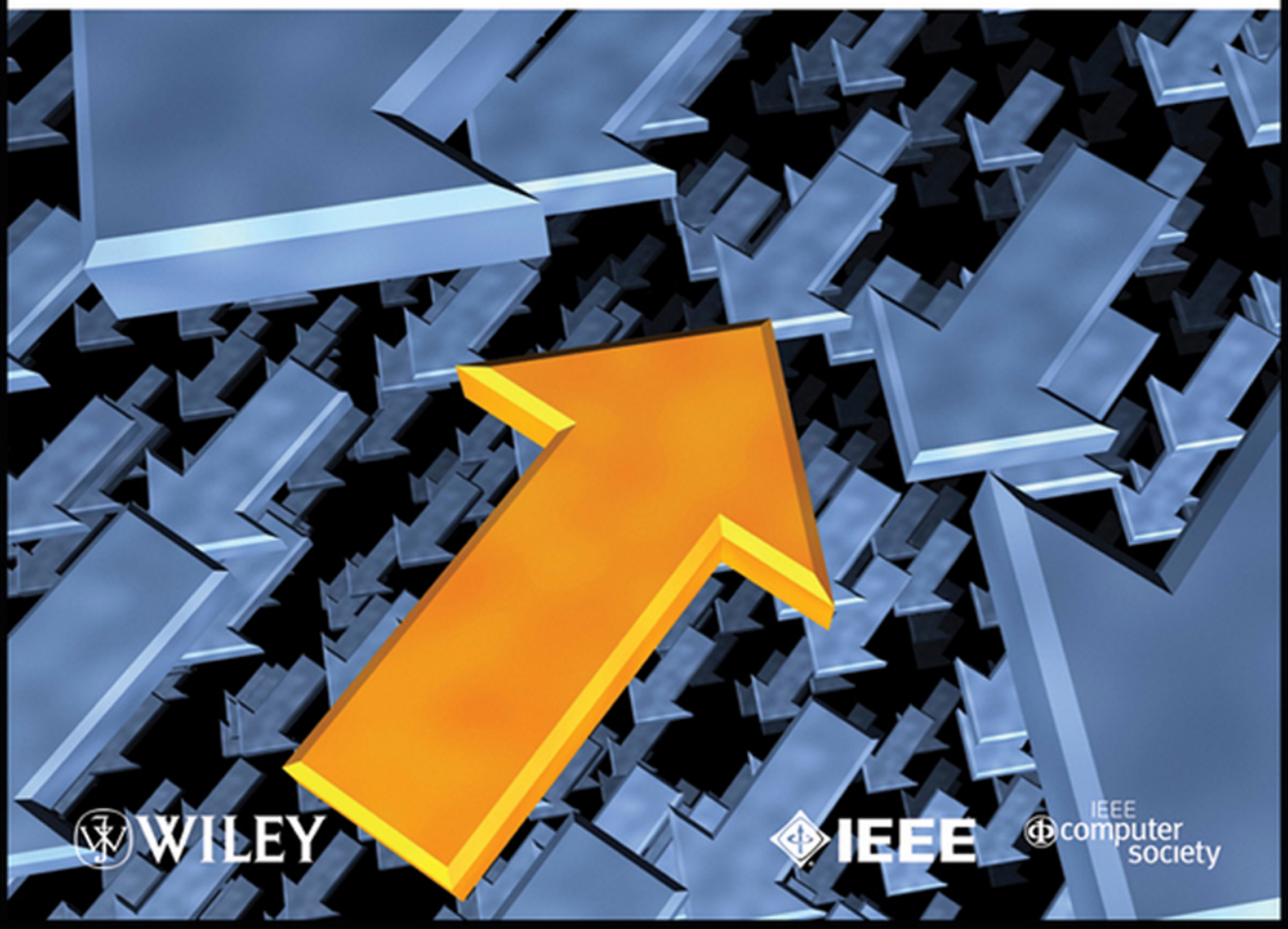
WWW. LINK AVAILABLE

Compiler Construction

Using Java™, JavaCC, and Yacc

Anthony J. Dos Reis



Compiler Construction Using Java, JavaCC, and Yacc

Compiler Construction Using Java, JavaCC, and Yacc

ANTHONY J. DOS REIS
State University of New York at New Paltz

IEEE computer society

WILEY

A JOHN WILEY & SONS, INC., PUBLICATION

Library of Congress Cataloging-in-Publication Data:

Dos Reis, Anthony J.
Compiler construction using Java, JavaCC, and Yacc / Anthony J. Dos Reis.
p. cm.
ISBN 978-0-470-94959-7 (hardback)
1. Compilers (Computer programs) 2. Java (Computer program language) I. Title.
QA76.76.C65D67 2011
005.4'53—dc23 2011013211

Printed in the United States of America.

oBook ISBN: 978-1-118-11276-2
ePDF ISBN: 978-1-118-11287-8
ePub ISBN: 978-1-118-11277-9
eMobi ISBN: 978-1-118-11288-5

10 9 8 7 6 5 4 3 2 1

To little sister

CONTENTS

Preface xv

Chapter 1 Strings, Languages, and Compilers 1

1.1 Introduction 1
1.2 Basic Language Concepts 1
1.3 Basic Compiler Concepts 3
1.4 Basic Set Theory 4
1.5 Null String 6
1.6 Concatenation 7
1.7 Exponent Notation 7
1.8 Star Operator 8
1.9 Concatenation of Sets of Strings 9
1.10 Plus Operator 11
1.11 Question Mark Operator 11
1.12 Shorthand Notation for a Set Containing a Single String 12
1.13 Operator Precedence 12
1.14 Regular Expressions 13
1.15 Limitations of Regular Expressions 15
Problems 16

Chapter 2 Context-Free Grammars, Part 1 19

2.1 Introduction 19
2.2 What is a Context-Free Grammar? 20
2.3 Derivations Using a Context-Free Grammar 21
2.4 Language Defined by a Context-Free Grammar 23
2.5 Different Ways of Representing Contet-Free Grammars 25
2.6 Some Simple Grammars 26
2.7 Techniques for Generating Languages with Context-Free Grammars 29
2.8 Regular and Right Linear Grammars 35

2.9	Counting with Regular Grammars	37
2.0	Grammars for Lists	39
2.10	An Important Language that is Not Context Free	44
	Problems	45
Chapter 3	**Context-Free Grammars, Part 2**	**49**
3.1	Introduction	49
3.2	Parse Trees	49
3.3	Leftmost and Rightmost Derivations	51
3.4	Substitution	52
3.5	Ambiguous Grammars	54
3.6	Determining Nullable Nonterminals	59
3.7	Eliminating Lambda Productions	60
3.8	Eliminating Unit Productions	64
3.9	Eliminating Useless Nonterminals	66
3.10	Recursion Conversions	71
3.11	Adding the Null String to a Language	76
	Problems	77
Chapter 4	**Context-Free Grammars, Part 3**	**83**
4.1	Introduction	83
4.2	Grammars for Arithmetic Expressions	83
4.3	Specifying Associativity and Precedence in Grammars	90
4.4	Backus–Naur Form	92
4.5	Syntax Diagrams	94
4.6	Abstract Syntax Trees and Three-Address Code	96
4.7	Noncontracting Grammars	97
4.8	Essentially Noncontracting Grammars	97
4.9	Converting a Context-Free Grammar to an Essentially Noncontracting Grammar	98
4.10	Pumping Property of Context-Free Languages	101
	Problems	104
Chapter 5	**Chomsky's Hierarchy**	**107**
5.1	Introduction	107
5.2	Context-Sensitive Productions	107
5.3	Context-Sensitive Grammars	110
5.4	Unrestricted Grammars	111
	Problems	112
Chapter 6	**Top-Down Parsing**	**115**
6.1	Introduction	115
6.2	Top-Down Construction of a Parse Tree	115
6.3	Parses that Fail	117
6.4	A Bad Grammar for Top-Down Parsing	118
6.5	Deterministic Parsers	119
6.6	A Parser that Uses a Stack	120
6.7	Table Representation of a Stack Parser	124
6.8	Handling Productions with Nonleading Terminal	126

6.9 Writing a Stack Parser in Java 127
Problems 134

Chapter 7 LL(1) Grammars 137

7.1 Introduction 137
7.2 FIRST Set of the Right Side of a Production 137
7.3 Determining Operation Sequences 140
7.4 Determining Selection Sets of Lambda Productions 142
7.5 Whatever-Follows-Left-Follows-Rightmost Rule 145
7.6 Selection Sets for Productions with Nullable Right Sides 147
7.7 Selection Sets Containing End-of-Input Symbol 149
7.8 A Stack Parser for a Grammar with Lambda Productions 152
7.9 Converting a Non-LL(1) Grammar to an LL(1) Grammar 153
7.10 Parsing with an Ambiguous Grammar 160
7.11 Computing FIRST and FOLLOW Sets 163
Problems 165

Chapter 8 Table-Driven Stack Parser 171

8.1 Introduction 171
8.2 Unifying the Operations of a Stack Parser 172
8.3 Implementing a Table-Driven Stack Parser 175
8.4 Improving Our Table-Driven Stack Parser 180
8.5 Parsers that are Not Deterministic—A Digression on Theory 181
Problems 183

Chapter 9 Recursive-Descent Parsing 185

9.1 Introduction 185
9.2 Simple Recursive-Descent Parser 185
9.3 Handling Lambda Productions 192
9.4 A Common Error 197
9.5 Java Code for Productions 198
9.6 Left Factoring in a Recursive-Descent Parser 199
9.7 Eliminating Tail Recursion 204
9.8 Translating the Star, Plus, and Question Mark Operators 108
9.9 Doing Things Backward 210
Problems 211

Chapter 10 Recursive-Descent Translation 215

10.1 Introduction 215
10.2 A Simple Translation Grammar 215
10.3 Converting a Translation Grammar to Java Code 217
10.4 Specifications for a Translation Grammar 218
10.5 Passing Information During a Parse 231
10.6 L-Attributed Grammars 236
10.7 New Token Manager 238
10.8 Solving the Token Lookahead Problem 241
10.9 Code for the New Token Manager 241
10.10 Translation Grammar for Prefix Expression Compiler 253

10.11 An Interesting Use of Recursion 257
Problems 261

Chapter 11 Assembly Language 265

11.1 Introduction 265
11.2 Structure of the J1 Computer 265
11.3 Machine Language Instructions 266
11.4 Assembly Language Instructions 268
11.5 Pushing Characters 269
11.6 `aout` Instruction 270
11.7 Using Labels 270
11.8 Using the Assembler 272
11.9 `stav` Instruction 275
11.10 Compiling an Assignment Statement 277
11.11 Compiling `print` and `println` 280
11.12 Outputting Strings 281
11.13 Inputting Decimal Numbers 283
11.14 `Entry` Directive 284
11.15 More Assembly Language 285
Problems 285

Chapter 12 S1—A Simple Compiler 289

12.1 Introduction 289
12.2 The Source Language 289
12.3 Grammar for Source Language 290
12.4 The Target Language 291
12.5 Symbol Table 292
12.6 Code Generator 293
12.7 `Token` Class 293
12.8 Writing the Translation Grammar 294
12.9 Implementing the S1 Compiler 299
12.10 Trying Out S1 315
12.11 Advice on Extending the S1 Compiler 318
12.12 Specifications for S2 320
Problems 324

Chapter 13 JavaCC 331

13.1 Introduction 331
13.2 JavaCC Extended Regular Expressions 333
13.3 JavaCC Input File 337
13.4 Specifying Actions for Regular Expressions 344
13.5 JavaCC Input File for S1j 348
13.6 Files Produced by JavaCC 355
13.7 Using the Star and Plus Operators 359
13.8 Choice Points and the Lookahead Directive 362
13.9 JavaCC's Choice Algorithm 367
13.10 Syntactic and Semantic Lookahead 371
13.11 Using JavaCC to Create a Token Manager Only 372
13.12 Using the Token Chain 373
13.13 Suppressing Warning Messages 377
Problems 387

Chapter 14 Building on S2 **383**

14.1 Introduction 383
14.2 Extending `println` and `print` 383
14.3 Cascaded Assignment Statement 388
14.4 Unary Plus and Minus 313
14.5 `readint` Statement 393
14.6 Controlling the Token Trace from the Command Line 395
14.7 Specifications for S3 396
Problems 396

Chapter 15 Compiling Control Structures **399**

15.1 Introduction 399
15.2 `while` Statement 399
15.3 `if` Statement 403
15.4 `do-while` Statement 407
15.5 Range Checking of Numerical Constants 408
15.6 Handling Backslash-Quote in a String 410
15.7 Handling Backslash-Quote with JavaCC 411
15.8 Universal Blocks in JavaCC 416
15.9 Handling Strings that Span Lines 418
15.10 Handling Strings that Span Lines Using JavaCC 419
15.11 `SPECIAL_TOKEN` Block in JavaCC 422
15.12 Error Recovery 424
15.13 Error Recovery in JavaCC 429
15.14 Specifications for S4 430
Problems 431

Chapter 16 Compiling Programs in Functional Form **435**

16.1 Introduction 435
16.2 Separate Assembly and Linking 435
16.3 Calling and Returning from Fuctions 439
16.4 Source Language for S5 443
16.5 Symbol Table for S5 445
16.6 Code Generator for S5 446
16.7 Translation Grammar for S5 447
16.8 Linking with a Library 457
16.9 Specifications for S5 458
16.10 Extending S5 458
Problems 461

Chapter 17 Finite Automata **465**

17.1 Introduction 465
17.2 Deterministic Finite Automata 466
17.3 Converting a DFA to a Regular Expression 468
17.4 Java Code for a DFA 472
17.5 Nondeterministic Finite Automata 474
17.6 Using an NFA as an Algorithm 476
17.7 Converting an NFA to a DFA with the Subset Algorithm 478
17.8 Converting a DFA to a Regular Grammar 479
17.9 Converting a Regular Grammar to an NFA 482

17.10 Converting a Regular Expression to an NFA 484
17.11 Finding the Minimal DFA 488
17.12 Pumping Property of Regular Languages 493
Problems 495

Chapter 18 Capstone Project: Implementing Grep Using Compiler Technology 499

18.1 Introduction 499
18.2 Regular Expressions for Our Grep Program 501
18.3 Token Manager for Regular Expression 501
18.4 Grammar for Regular Expressions 503
18.5 Target Language for Our Regular Expression Compiler 503
18.6 Using an NFA for Pattern Matching 508
Problems 513

Chapter 19 Compiling to a Register-Oriented Architecture 515

19.1 Introduction 515
19.2 Using the Register Instruction Set 516
19.3 Modifications to the Symbol Table for R1 517
19.4 Parser and Code Generator for R1 518
Problems 526

Chapter 20 Optimization 529

20.1 Introduction 529
20.2 Using the `ldc` Instruction 531
20.3 Reusing Temporary Variables 532
20.4 Constant Folding 535
20.5 Register Allocation 537
20.6 Peephole Optimization 540
Problems 543

Chapter 21 Interpreters 547

21.1 Introduction 547
21.2 Converting S1 to I1 549
21.3 Interpreting Statements that Transfer Control 552
21.4 Implementing the Compiler-Interpreter CI1 553
21.5 Advantages of Interpreters 558
Problems 559

Chapter 22 Bottom-Up Parsing 561

22.1 Introduction 561
22.2 Principles of Bottom-Up Parsing 561
22.3 Parsing with Right- versus Left-Recursive Grammars 565
22.4 Bottom-Up Parsing with an Ambiguous Grammar 566
22.5 Do-Not-Reduce Rule 569
22.6 SLR(1) Parsing 570
22.7 Shift/Reduce Conflicts 577
22.8 Reduce/Reduce Conflicts 579
22.9 LR(1) Parsing 579
Problems 584

Chapter 23 yacc **587**

23.1 Introduction 587
23.2 yacc Input and Output Files 587
23.3 A Simple yacc-Generated Parser 588
23.4 Passing Values Using the Value Stack 596
23.5 Using yacc With an Ambiguous Grammar 602
23.6 Passing Values down the Parse Tree 604
23.7 Implementing S1y 606
23.8 jflex 612
Problems 618

Appendix A Stack Instruction Set **621**

Appendix B Register Instruction Set **625**

References **629**

Index **631**

PREFACE

My principal goal in writing this book is to provide the reader with a clear exposition of the theory of compiler design and implementation along with plenty of opportunities to put that theory into practice. The theory, of course, is essential, but so is the practice.

NOTABLE FEATURES

- Provides numerous, well-defined projects along with test cases. These projects ensure that students not only know the theory but also know how to apply it. Instructors are relieved of the burden of designing projects that integrate well with the text.
- Project work starts early in the book so that students can apply the theory as they are learning it.
- The compiler tools (JavaCC, Yacc, and Lex) are optional topics.
- The entire book is Java oriented. The implementation language is Java. The principal target language is similar to Java's bytecode. The compiler tools generate Java code. The form of attributed grammar used has a Java-like syntax.
- The target languages (one is stack oriented like Java's bytecode; the other is register oriented) are very easy to learn but are sufficiently powerful to support advanced compiler projects.
- The software package is a dream come true for both students and instructors. It automatically evaluates a student's compiler projects with respect to correctness, run time, and size. It is great for students: they get immediate feedback on their projects. It is great for instructors: they can easily and accurately evaluate a student's work. With a single command, an instructor can generate a report for an entire class. The software runs on three platforms: Microsoft Windows, Linux, and the Macintosh OS X.
- Demonstrates how compiler technology is not just for compilers. In a capstone project, students design and implement grep using compiler technology.
- Includes a chapter on interpreters that fits in with the rest of the book.

- Includes a chapter on optimization that is just right for an introductory course. Students do not simply read about optimization techniques—they implement a variety of techniques, such as constant folding, peephole optimization, and register allocation.
- The book uses a Java-like form of grammars that students can easily understand and use. This is the same form that JavaCC uses. Thus, students can make transition to JavaCC quickly and easily.
- Provides enough theory that the book can be used for a combined compiler/automata/formal languages course. The book covers most of the topics covered in an automata/formal languages course: finite automata, stack parsers, regular expressions, regular grammars, context-free grammars, context-sensitive grammars, unrestricted grammars, Chomsky's hierarchy, and the pumping lemmas. Pushdown automata, Turing machines, computability, and complexity are discussed in supplements in the software package. The software package also includes a pushdown automaton simulator and Turing machine simulator.
- Covers every topic that should be in a first course or in an only course on compilers. Students will learn not only the theory and practice of compiler design but also important system concepts.

SOFTWARE PACKAGE

The software package for the textbook has some unusual features. When students run one of their compiler-generated programs, the software produces a log file. The log file contains a time stamp, the student's name, the output produced by the compiler-generated program, and an evaluation of the compiler-generated program with respect to correctness, program size, and execution time. If the output is not correct (indicating that the student's compiler is generating incorrect code), the log file is marked with NOT CORRECT. If the compiled program is too big or the execution time too long, the log file is marked with OVER LIMIT.

The name of a log file contains the student's name. For example, the log file for the S3 project of a student whose last name is Dos Reis would be S3.dosreis.log. Because each log file name is unique, an instructor can store all the log files for a class in a single directory. A single command will then produce a report for the entire class.

The software supports two instruction sets: the stack instruction set and the register instruction set. The stack instruction set is the default instruction set. To use the register instruction set, a single directive is placed in the assembly language source program. The software then automatically reconfigures itself to use the register instruction set.

The three principal programs in the software package are **a** (the assembler/linker), **e** (the executor), and **l** (the library maker). The software package also includes **p** (a pushdown automaton simulator) and **t** (a Turing machine simulator).

The software package for this book is available from the publisher. The compiler tools are available on the Web. At the time of this writing, JavaCC is at http://java.net/downloads/javacc, Byacc/j is at http://byaccj.sourceforge.net/, and Jflex is at http://jflex.de/.

PROJECTS

This textbook specifies many well-defined projects. The source language has six levels of increasing complexity. A student can write a compiler for each level that translates to the

stack instruction set. A student can also write a compiler for each level that translates to the register instruction set, or incorporates optimization techniques. For each level, a student can write a pure interpreter or an interpreter that uses an intermediate code. A student can implement several variations of grep using compiler technology. A student can write the code for any of these projects by hand or by using JavaCC or Yacc. Many of the chapter problems provide additional projects. In short, there are plenty of projects.

For each project, the textbook provides substantial support. Moreover, many of the projects are incremental enhancements of a previous project. This incremental approach works well; each project is challenging but not so challenging that students cannot do it.

Most projects can be completed in a week's time. Thus, students should be able to do ten or even more projects in a single semester.

USEFUL REFERENCES

For background material, the reader is referred to the author's *An Introductions to Programming Using Java* (Jones & Bartlett, 2010) and *Assembly Language and Computer Architecture Using C++ and Java* (Course Technology, 2004). Also recommended is JFLAP (available at http://www.jflap.org), an interactive program that permits experimentation with various types of automata and grammars.

ACKNOWLEDGMENTS

I would like to thank Professors Robert McNaughton and Dean Arden who years ago at RPI taught me the beauty of formal language theory, my students who used preliminary versions of the book and provided valuable feedback, Katherine Guillemette for her support of this project, my daughter Laura for her suggestions on the content and structure of the book, and my wife for her encouragement.

ANTHONY J. DOS REIS

New Paltz, New York
October 2011

1

STRINGS, LANGUAGES, AND COMPILERS

1.1 INTRODUCTION

Compiler construction is truly an engineering science. With this science, we can methodically—almost routinely—design and implement fast, reliable, and powerful compilers.

You should study compiler construction for several reasons:

- Compiler construction techniques have very broad applicability. The usefulness of these techniques is not limited to compilers.
- To program most effectively, you need to understand the compiling process.
- Language and language translation are at the very heart of computing. You should be familiar with their theory and practice.
- Unlike some areas of computer science, you do not typically pick up compiler construction techniques "on the job." Thus, the formal study of these techniques is essential.

To be fair, you should also consider reasons for *not* studying compiler construction. Only one comes to mind: Your doctor has ordered you to avoid excitement.

1.2 BASIC LANGUAGE CONCEPTS

In our study of compiler design theory, we begin with several important definitions. An *alphabet* is the finite set of characters used in the writing of a language. For example, the alphabet of the Java programming language consists of all the characters that can appear in a program: the upper- and lower-case letters, the digits, whitespace (space, tab, newline, and carriage return), and all the special symbols, such as =, +, and {. For most of the examples in this book, we will use very small alphabets, such as {b, c} and {b, c, d}. We

Compiler Construction Using Java, JavaCC, and Yacc, First Edition. Anthony J. Dos Reis

will avoid using the letter "a" in our alphabets because of the potential confusion with the English article "a".

A *string over an alphabet* is a finite sequence of characters selected from that alphabet. For example, suppose our alphabet is {b, c, d}. Then

```
cbd
cbcc
c
```

are examples of strings over our alphabet. Notice that in a string over an alphabet, each character in the alphabet can appear any number of times (including zero times) and in any order. For example, in the string cbcc (a string over the three-letter alphabet {b, c, d}), the character b appears once, c appears three times, and d does not appear.

The *length of a string* is the number of characters the string contains. We will enclose a string with vertical bars to designate its length. For example, |cbcc| designates the length of the string cbcc. Thus, |cbcc| = 4.

A *language* is a set of strings over some alphabet. For example, the set containing just the three strings cbd, cbcc, and c is a language. This set is not a very interesting language, but it is, nevertheless, a language according to our definition.

Let us see how our definitions apply to a "real" language—the programming language Java. Consider a Java program all written on a single line:

```
class C { public static void main(String[] args) {} }
```

Clearly, such a program is a single string over the alphabet of Java. We can also view a multiple-line program as a single string—namely, the string that is formed by connecting successive lines with a line separator, such as a newline character or a carriage return/newline sequence. Indeed, a multiline program stored in a computer file is represented by just such a string. Thus, the multiple-line program

```
class C
{
  public static void main(String[] args)
  {
  }
}
```

is the single string

```
class C□{□ public static void main(String[] args)□ {□ }□}
```

where □ represents the line separator. The *Java language* is the set of all strings over the Java alphabet that are valid Java programs.

A language can be either finite or infinite and may or may not have a meaning associated with each string. The Java language is infinite and has a meaning associated with each string. The meaning of each string in the Java language is what it tells the computer to do. In contrast, the language {cbd, cbcc, c} is finite and has no meaning associated with each string. Nevertheless, we still consider it a language. A language is simply a set, finite or infinite, of strings, each of which may or may not have an associated meaning.

Syntax rules are rules that define the form of the language, that is, they specify which strings are in a language. *Semantic rules* are rules that associate a meaning to each string in a language, and are optional under our definition of language.

Occasionally, we will want to represent a string with a single symbol very much like x is used to represent a number in algebra. For this purpose, we will use the small letters at the end of the English alphabet. For example, we might use x to represent the string `cbd` and y to represent the string `cbcc`.

1.3 BASIC COMPILER CONCEPTS

A *compiler* is a translator. It typically translates a program (the *source program*) written in one language to an equivalent program (the *target program*) written in another language (see Figure 1.1). We call the languages in which the source and target programs are written the *source* and *target languages*, respectively.

Typically, the source language is a high-level language in which humans can program comfortably (such as Java or C++), whereas the target language is the language the computer hardware can directly handle (*machine language*) or a symbolic form of it (*assembly language*).

If the source program violates a syntax rule of the source language, we say it has a *syntax error*. For example, the following Java method has one syntax error (a right brace instead of a left brace on the second line):

```
public void greetings()
}                                  // syntax error
  System.out.println("hello");
}
```

A *logic error* is an error that does not violate a syntax rule but results in the computer performing incorrectly when we run the program. For example, suppose we write the following Java method to compute and return the sum of 2 and 3:

```
public int sum()
{
  return 2 + 30;     // logic error
}
```

This method is a valid Java method but it tells the computer to do the wrong thing—to compute 2 + 30 instead 2 + 3. Thus, the error here is a logic error.

A compiler in its simplest form consists of three parts: the token manager, the parser, and the code generator (see Fig. 1.2).

The source program that the compiler inputs is a stream of characters. The *token manager* breaks up this stream into meaningful units, called *tokens*. For example, if a token manager reads

Figure 1.1.

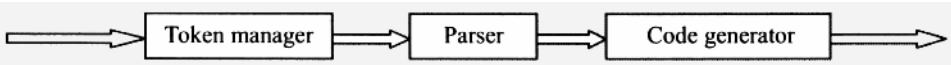


Figure 1.2.

```
int x;          // important example
x = 55;
```

it would output the following sequence of tokens:

```
int
x
;
x
=
55
;
```

The token manager does not produce tokens for white space (i.e., space, tab, newline, and carriage return) and comments because the parser does need these components of the source program. A token manager is sometimes called a *lexical analyzer, lexer, scanner,* or *tokenizer.*

A *parser* in its simplest form has three functions:

1. It analyzes the structure of the token sequence produced by the token manager. If it detects a syntax error, it takes the appropriate action (such as generating an error message and terminating the compile).
2. It derives and accumulates information from the token sequence that will be needed by the code generator.
3. It invokes the code generator, passing it the information it has accumulated.

The *code generator,* the last module of a compiler, outputs the target program based on the information provided by the parser.

In the compilers we will build, the parser acts as the controller. As it executes, it calls the token manager whenever it needs a token, and it calls the code generator at various points during the parse, passing the code generator the information the code generator needs. Thus, the three parts of the compiler operate concurrently. An alternate approach is to organize the compiling process into a sequence of *passes.* Each pass reads an input file and creates an output file that becomes the input file for the next pass. For example, we can organize our simple compiler into three passes. In the first pass, the token manager reads the source program and creates a file containing the tokens corresponding to the source program. In the second pass, the parser reads the file of tokens and outputs a file containing information required by the code generator. In the third pass, the code generator reads this file and outputs a file containing the target program.

1.4 BASIC SET THEORY

Since languages are sets of strings, it is appropriate at this point to review some basic set theory. One method of representing a set is simply to list its elements in any order. Typi-

cally, we use the left and right braces, "{" and "}", to delimit the beginning and end, respectively, of the list of elements. For example, we represent the set consisting of the integers 3 and 421 with

$\{3, 421\}$ or $\{421, 3\}$

Similarly, we represent the set consisting of the two strings `b` and `bc` with

`{b, bc}` or `{bc, b}`

This approach cannot work for an infinite set because it is, of course, impossible to list all the elements of an infinite set. If, however, the elements of an infinite set follow some obvious pattern, we can represent the set by listing just the first few elements, followed by the ellipsis (. . .). For example, the set

`{b, bb, bbb, ...}`

represents the infinite set of strings containing one or more `b`'s and no other characters. Representing infinite sets this way, however, is somewhat imprecise because it requires the reader to figure out the pattern represented by the first few elements.

Another method for representing a set—one that works for both finite and infinite sets—is to give a rule for determining its elements. In this method, a set definition has the form

$\{E : \textit{defining rule}\}$

where E is an expression containing one or more variables, and the defining rule generally specifies the allowable ranges of the variables in E. The colon means "such that." We call this representation the *set-builder notation.* For example, we can represent the set containing the integers 1 to 100 with

$\{x : x \text{ is an integer and } 1 \leq x \leq 100\}$

Read this definition as "the set of all x such that x is an integer and x is greater than or equal to 1 and less than or equal to 100." A slightly more complicated example is

$\{n^2 : n \text{ is an integer and } n \geq 1\}$

Notice that the expression preceding the colon is not a single variable as in the preceding example. The defining rule indicates that n can be 1, 2, 3, 4, and so on. The corresponding values of n^2 are the elements of the set—namely, 1, 4, 9, 16, etc. Thus, this is the infinite set of integer squares:

$\{1, 4, 9, 16, \ldots\}$

In set notation, the mathematical symbol $\in$ means "is an element of." A superimposed slash on a symbol negates the condition represented. Thus, $\notin$ means "is not a element of." For example, if $P = \{2, 3, 4\}$, then $3 \in P$, but $5 \notin P$.

The *empty set* [denoted by either $\{\}$ or ϕ] is the set that contains no elements. The *universal set* (denoted by U) is the set of all elements under consideration. For example, if we are working with sets of integers, then the set of all integers is our universe. If we are

working with strings over the alphabet {b, c}, then the set of all strings over {b, c} is our universe.

The set operations *union, intersection,* and *complement,* form new sets from given sets. The union operator is most often denoted by the special symbol $\cup$. We, however, use the vertical bar | to denote the union operator. The advantage of | is that it is available on standard keyboards. We will use $\cap$ and ~ to denote the intersection and complement operators, respectively. $\cap$, the standard symbol for set intersection, unfortunately is not available on keyboards. However, we will use set intersection so infrequently that it will not be necessary to substitute a keyboard character for $\cap$.

Set union, intersection, and complement are defined as follows:

Union of P and Q: $P \mid Q = \{x : x \in P \text{ or } x \in Q\}$
Intersection of P and Q: $P \cap Q = \{x : x \in P \text{ and } x \in Q\}$
Complement of P: $\sim P = \{x : x \in U \text{ and } x \notin P\}$

Here are the definitions in words of these operators:

$P \mid Q$ is the set of all elements that are in either P or Q or both.
$P \cap Q$ is the set of elements that are in both P and Q.
$\sim P$ is the set of all elements in the universe U that are not in P.

For example, if P = {b, bb}, Q = {bb, bbb}, and our universe U = {b, bb, bbb, . . .}, then

$P \mid Q$ = {b, bb, bbb}
$P \cap Q$ = {bb}
$\sim P$ = {bbb, bbbb, bbbbb, . . .}
$\sim Q$ = {b, bbbb, bbbbb, . . .}

A collection of sets is *disjoint* if the intersection of every pair of sets from the collection is the empty set (i.e., they have no elements in common). For example, the sets {b}, {bb, bbb}, and {bbbb} are disjoint since no two have any elements in common.

The set P is a subset of Q (denoted $P \subseteq Q$) if every element of P is also in Q. The set P is a proper subset of the set Q (denoted $P \subset Q$) if P is a subset of Q, and Q has at least one element not in P. For example, if P = {b, bb}, Q = {b, bb, bbb}, and R = {b, bb}, then P is proper subset of Q, but P is not a proper subset of R. However, P is a subset of R. Two sets are equal if each is the subset of the other. With P and R given as above, $P \subseteq R$ and $R \subseteq P$. So we can conclude that $P = R$. Note that the empty set is a subset of any set; that is, $\{\} \subseteq S$ for any set S.

We can apply the set operations union, intersection, and complement to any sets. We will soon see some additional set operations specifically for sets of strings.

1.5 NULL STRING

When prehistoric humans started using numbers, they used the natural numbers 1, 2, 3, ... It was easy to grasp the idea of oneness, twoness, threeness, and so on. Therefore, it was natural to have symbols designating these concepts. In contrast, the number 0 is hardly a natural concept. After all, how could something (the symbol 0) designate nothing? Today,

of course, we are all quite comfortable with the number 0 and put it to good use every day. A similar situation applies to strings. It is natural to think of a string as a sequence of one or more characters. But, just as the concept zero is useful to arithmetic, so is the concept of a *null string*—the string whose length is zero—useful to language theory. The null string is the string that does not contain any characters.

How do we designate the null string? Normally, we designate strings by writing them down on a piece of paper. For example, to designate a string consisting of the first three small letters of the English alphabet, we write `abc`. A null string, however, does not have any characters, so there is nothing to write down. We need some symbol, preferably one that does not appear in the alphabets we use, to represent the null string. Some writers of compiler books use the Greek letter ϵ for the null string. However, since ϵ is easily confused with the symbol for set membership, we will use the small Greek letter λ (lambda) to represent the null string.

One common misconception about the null string is that a string consisting of a single space is the null string. A space is a character whose length is one; the null string has length zero. They are not the same. Another misconception has to do with the empty set. The null string is a string. Thus, the set $\{\lambda\}$ contains exactly one string—namely the null string. The empty set $\{\}$, on the other hand, does not contain any string.

1.6 CONCATENATION

We call the operation of taking one string and placing it next to another string in the order given to form a new string *concatenation*. For example, if we concatenate `bcd` and `efg`, we get the string `bcdefg`. Note that the concatenation of any string x with the null string λ yields x. That is,

$$x\lambda = \lambda x = x$$

1.7 EXPONENT NOTATION

A nonnegative exponent applied to a character or a sequence of characters in a string specifies the replication of that character or sequence of characters. For example $\mathtt{b}^4$ is a shorthand representation of `bbbb`. We use parentheses if the scope of the replication is more than one character. Hence, $\mathtt{b(cd)}^2\mathtt{e}$ represents `bcdcde`. A string replicated zero times is by definition the null string; that is, for any string x, $x^0 = \lambda$.

We can use exponent notation along with set-builder notation to define sets of strings. For example, the set

$$\{\mathtt{b}^i : 1 \leq i \leq 3\}$$

is the set

$$\{\mathtt{b}^1, \mathtt{b}^2, \mathtt{b}^3\} = \{\mathtt{b}, \mathtt{bb}, \mathtt{bbb}\}$$

The exponent in exponent notation can never be less than zero. If we do not specify its lower bound in a set definition, assume it is zero. For example, the set

$$\{\mathtt{b}^i : i \leq 3\}$$

should be interpreted as

$$\{b^i : 0 \leq i \leq 3\} = \{b^0, b^1, b^2, b^3\} = \{\lambda, b, bb, bbb\}$$

Exercise 1.1

Describe in English the language defined by $\{b^i c^{2i} : i \geq 0\}$.

Answer:

The set of all strings consisting of b's followed by c's in which the number of c's is twice the number of b's. This set is $\{\lambda, bcc, bbcccc, bbbcccccc, \ldots\}$.

■

1.8 STAR OPERATOR (ALSO KNOWN AS THE ZERO-OR-MORE OPERATOR)

We have just seen that an exponent following a character represents a single string (for example, b^3 represents bbb). In contrast, the star operator, *, following a character (for example, b*) represents a set of strings. The set contains every possible replication (including zero replications) of the starred character. For example,

$$b^* = \{b^0, b^1, b^2, b^3, \ldots\} = \{b^n : n \geq 0\} = \{\lambda, b, bb, bbb, \ldots\}$$

Think of the star operator as meaning "zero or more."

The star operator always applies to the item immediately preceding it. If a parenthesized expression precedes the star operator, then the star applies to whatever is inside the parentheses. For example, in (bcd)*, the parentheses indicate that the star operation applies to the entire string bcd. That is,

$$(bcd)^* = \{\lambda, bcd, bcdbcd, bcdbcdbcd, \ldots\}$$

The star operator can also be applied to sets of strings. If A is a set of strings, then A^* is the set of strings that can be formed from the strings of A using concatenation, allowing any string in A to be replicated any number of times (including zero times) and used in any order. By definition, the null string is always in A^*.

Here are several examples of starred sets:

$\{b\}^* = \{\lambda, b, bb, bbb, \ldots\} = b^*$
$\{b, c\}^* = \{\lambda, b, c, bb, bc, cb, cc, bbb, \ldots\}$
$\{\lambda\}^* = \{\lambda\}$
$\{\}^* = \{\lambda\}$
$\{bb, cc\}^* = \{\lambda, bb, cc, bbbb, bbcc, ccbb, cccc, \ldots\}$
$\{b, cc\}^* = \{\lambda, b, bb, cc, bbb, bcc, ccb, bbbb \ldots\}$

Notice that $\{b\}^* = b^*$. That is, starring a set that contains just one string yields the same set as starring just that string.

Here is how to determine if a given string is in A^*, where A is an arbitrary set of strings: If the given string is the null string, then it is in A^* by definition. If the given string is nonnull, and it can be divided into substrings such that each substring is in A,

then the given string is in A^*. Otherwise, the string is not in A^*. For example, suppose A = {b, cc}. We can divide the string bccbb into four parts: b, cc, b, and b, each of which is in A. Therefore, bccbb $\in A^*$. On the other hand, for the string bccc the required subdivision is impossible. If we divide bccc into b, cc, and c, the first two strings are in A but the last is not. All other subdivisions of bccc similarly fail. Therefore, bccc $\notin A^*$.

We call the set that results from the application of the star operator to a string or set of strings the *Kleene closure,* in honor of Stephen C. Kleene, a pioneer in theoretical computer science.

Let us now use the star operator to restate two important definitions that we gave earlier. Let the capital Greek letter Σ (sigma) represent an arbitrary alphabet. A *string over the alphabet* Σ is any string in Σ^*. For example, suppose Σ = {b, c}. Then

$$\Sigma^* = \{\lambda, \text{b}, \text{c}, \text{bb}, \text{bc}, \text{cb}, \text{cc}, \text{bbb}, \ldots\}$$

Thus, λ, b, c, bb, bc, cb, cc, bbb,. . . are strings over Σ. It may appear strange to view λ as a string over the alphabet Σ = {b, c}. Actually, this view is quite reasonable since λ has no characters *not* in {b, c}. λ is always a string over Σ regardless of the content of Σ because, by definition, λ is always in Σ^*. A *language over the alphabet* Σ is any subset of Σ^*. For example {λ}, {b} , and {b, cc} are each languages over Σ = {b, c}. Even the empty set is a language over Σ because it is a subset of Σ^*.

Exercise 1.2

a) List all the strings of length 3 in {b, cc}*.

b) Is ccbcc $\in$ {b, cc}*?

Answer:

a) bbb, bcc, ccb.

b) Yes. To confirm this, subdivide ccbcc into cc, b, and cc, all of which are elements of (b, cc}.

■

1.9 CONCATENATION OF SETS OF STRINGS

Concatenation can be applied to sets of strings as well as individual strings. If we let A and B be two sets of strings, then AB, the *concatenation of the sets A* and B, is

$$\{xy : x \in A \text{ and } y \in B\}$$

That is, AB is the set of all strings that can be formed by concatenating a string A with a string B. For example, if A = {b, cc} and B = {d, dd}, then

AB = {bd, bdd, ccd, ccdd}
BA = {db, dcc, ddb, ddcc}

As an example of concatenation, consider the set b*c*, the concatenation of the sets b* and c*. Each string in b*c* consists of some string from b* concatenated to some

string in c*. That is, each string consists of zero or more b's followed by zero or more c's. The number of b's does not have to equal the number of c's, but all b's must precede all c's. Thus, b*c* = {λ, b, c, bb, bc, cc, bbb, bbc, bcc, ccc, . . .}. In exponent notation, $\mathtt{b}^*\mathtt{c}^* = \{\mathtt{b}^i\mathtt{c}^j : i \geq 0 \text{ and } j \geq 0\}$.

A string can also be concatenated with a set. If x is a string and A is a set of strings, then xA, the concatenation of x with A is

$$\{xy : y \in A\}$$

Similarly, Ax is

$$\{yx : y \in A\}$$

For example, bbc*, the concatenation of the string bb and the set c*, is the set of all strings consisting of bb followed by a string in c*. Thus,

bbc* = {bbλ = bb, bbc, bbcc, bbccc, . . .}

Notice that it follows from our definitions that $xA = \{x\}A$, where x is an arbitrary string and A is a set of strings. That is, we get the same result whether we concatenate x (the string) or $\{x\}$ (the set containing just x) to a set A.

Exercise 1.3

a) List all strings in b*cb* of length less than 3.

b) Write an expression using the star operator which defines the same set as $\{\mathtt{b}^p\mathtt{c}^q\mathtt{d}^r : p \geq 0, q \geq 1, r \geq 2\}$.

Answer:

a) c, bc, cb.

b) b*cc*ddd*.

■

The union operator implies a choice with respect to the makeup of the strings in the language specified. For example, we can interpret

{b}({c} | {d})

as the set of strings consisting of a b followed by a choice of c or d. That is, the set consists of the strings bc and bd.

Exercise 1.4

Describe in English the set defined by b*({c} | {d})e*.

Answer:

The set of all strings consisting of zero or more b's, followed by either c or d, followed by zero or more e's.

■