Compiler Design: Theory, Tools, and Examples

Seth D. Bergmann

February 12, 2016

Contents

Preface

Compiler design is a subject which many believe to be fundamental and vital to computer science. It is a subject which has been studied intensively since the early 1950's and continues to be an important research field today. Compiler design is an important part of the undergraduate curriculum for many reasons: (1) It provides students with a better understanding of and appreciation for programming languages. (2) The techniques used in compilers can be used in other applications with command languages. (3) It provides motivation for the study of theoretic topics. (4) It is a good vehicle for an extended programming project.

There are several compiler design textbooks available today, but most have been written for graduate students. Here at Rowan University, our students have had difficulty reading these books. However, I felt it was not the subject matter that was the problem, but the way it was presented. I was sure that if concepts were presented at a slower pace, with sample problems and diagrams to illustrate the concepts, that our students would be able to master the concepts. This is what I have attempted to do in writing this book.

This book is a revision of earlier editions that were written for Pascal and C++ based curricula. As many computer science departments have moved to Java as the primary language in the undergraduate curriculum, I have produced this edition to accommodate those departments. This book is not intended to be strictly an object- oriented approach to compiler design. Though most Java compilers compile to an intermediate form known as Byte Code, the approach taken here is a more traditional one in which we compile to native code for a particular machine.

The most essential prerequisites for this book are courses in Java application programming, Data Structures, Assembly Language or Computer Architecture, and possibly Programming Languages. If the student has not studied formal languages and automata, this book includes introductory sections on these theoretic topics, but in this case it is not likely that all seven chapters will be covered in a one semester course. Students who have studied the theory will be able to skip the preliminary sections (2.0, 3.0, 4.0) without loss of continuity.

The concepts of compiler design are applied to a case study which is an implementation of a subset of Java which I call Decaf. Chapters 2, 4, 5, and 6 include a section devoted to explaining how the relevant part of the Decaf compiler is designed. This public domain software is presented in full in the

appendices and is available on the Internet. Students can benefit by enhancing or changing the Decaf compiler provided.

Chapters 6 and 7 focus on the back end of the compiler (code generation and optimization). Here I rely on a fictitious computer, called Mini, as the target machine. I use a fictitious machine for three reasons: (1) I can design it for simplicity so that the compiler design concepts are not obscured by architectural requirements, (2) It is available to anyone who has a C compiler (the Mini simulator, written in C, is available also), and (3) the teacher or student can modify the Mini machine to suit his/her tastes.

Chapter 7 includes only a brief description of optimization techniques since there is not enough time in a one semester course to delve into these topics, and because these are typically studied in more detail at the graduate level.

To use the software that accompanies this book, you will need access to the world wide web. The source files can be accessed at
`http://www.rowan.edu/~bergmann/books/java/decaf`.

These are plain text files which can be saved from your internet browser. Additional description of these files can be found in Appendix B.

I wish to acknowledge the people who participated in the design of this book. The reviewers of the original Pascal version. James E. Miller of Transylvania University, Jeffrey C. Chang of Garner-Webb University, Stephen J. Allan of Utah State University, Karsten Henckell of the New College of USF, and Keith Olson of Montana Technical College all took the time to read through various versions of the manuscript of the original edition and provided many helpful suggestions. My students in the Compiler Design course here at Rowan University also played an important role in testing the original version and subsequent versions of this book. Support in the form of time and equipment was provided by the administration of Rowan University.

The pages of this book were composed entirely by the authors and contributors using LaTeX (with extensions DraTeX and AlDraTeX). Finally, I am most grateful to my wife Sue for being so understanding during the time that I spent working on this project.

Secondary Authors

This book is the result of an attempt to launch a series of *open source* textbooks. Source files are available at
`http://cs.rowan.edu/~bergmann/books`.

Contributor List

If you have a suggestion or correction, please send email to bergmann@rowan.edu. If I make a change based on your feedback, I will add you to the contributor list (unless you ask to be omitted).

If you include at least part of the sentence the error appears in, that makes it easy for me to search. Page and section numbers are fine, too, but not quite as easy to work with.

If you wish to rewrite a section or chapter, it would be a good idea to notify me before starting on it. Major rewrites can qualify for "secondary author" status.

•

Chapter 1

Introduction

Recently the phrase user interface has received much attention in the computer industry. A user interface is the mechanism through which the user of a device communicates with the device. Since digital computers are programmed using a complex system of binary codes and memory addresses, we have developed sophisticated user interfaces, called programming languages, which enable us to specify computations in ways that seem more natural. This book will describe the implementation of this kind of interface, the rationale being that even if you never need to design or implement a programming language, the lessons learned here will still be valuable to you. You will be a better programmer as a result of understanding how programming languages are implemented, and you will have a greater appreciation for programming languages. In addition, the techniques which are presented here can be used in the construction of other user interfaces, such as the query language for a database management system.

1.1 What is a Compiler?

Recall from your study of assembly language or computer organization the kinds of instructions that the computer's CPU is capable of executing. In general, they are very simple, primitive operations. For example, there are often instructions which do the following kinds of operations: (1) add two numbers stored in memory, (2) move numbers from one location in memory to another, (3) move information between the CPU and memory. But there is certainly no single instruction capable of computing an arbitrary expression such as $\sqrt{(x-x0)^2+(x-x1)^2}$, and there is no way to do the following with a single instruction:

```
if (array6[loc]<MAX) sum = 0; else array6[loc] = 0;
```

These capabilities are implemented with a software translator, known as a compiler. The function of the compiler is to accept statements such as those above and translate them into sequences of machine language operations which,

if loaded into memory and executed, would carry out the intended computation. It is important to bear in mind that when processing a statement such as x = x * 9; the compiler does not perform the multiplication. The compiler generates, as output, a sequence of instructions, including a "multiply" instruction.

Languages which permit complex operations, such as the ones above, are called high-level languages, or programming languages. A compiler accepts as input a program written in a particular high-level language and produces as output an equivalent program in machine language for a particular machine called the target machine. We say that two programs are equivalent if they always produce the same output when given the same input. The input program is known as the source program, and its language is the source language. The output program is known as the object program, and its language is the object language. A compiler translates source language programs into equivalent object language programs. Some examples of compilers are:

A Java compiler for the Apple Macintosh

A COBOL compiler for the SUN

A C++ compiler for the Apple Macintosh

If a portion of the input to a Java compiler looked like this:

$a = b + c * d;$

the output corresponding to this input might look something like this:

```
LOD     r1,c            // Load the value of c into reg 1
MUL     r1,d            // Multiply the value of d by reg 1
STO     r1,temp1        // Store the result in temp1
LOD     r1,b            // Load the value of b into reg 1
ADD     r1,temp1        // Add value of temp1 to register 1
STO     r1,temp2        // Store the result in temp2
MOV     a,temp2         // Move temp2 to a, the final result
```

The compiler must be smart enough to know that the multiplication should be done before the addition even though the addition is read first when scanning the input. The compiler must also be smart enough to know whether the input is a correctly formed program (this is called checking for proper syntax), and to issue helpful error messages if there are syntax errors.

Note the somewhat convoluted logic after the Test instruction in Sample Problem 1.1.1 . Why did it not simply branch to L3 if the condition code indicated that the first operand (X) was greater than or equal to the second operand (temp1), thus eliminating an unnecessary branch instruction and label? Some compilers might actually do this, but the point is that even if the architecture of the target machine permits it, many compilers will not generate optimal code. In designing a compiler, the primary concern is that the object program be semantically equivalent to the source program (i.e. that they mean the same thing, or produce the same output for a given input). Object program efficiency is important, but not as important as correct code generation.

Sample Problem 1.1.1

Show the output of a Java native code compiler, in any typical assembly language, for the following Java input string:

```
while (x<a+b) x = 2*x;
```

Solution:

```
L1: LOD   r1,a          // Load a into reg. 1
    ADD   r1,b          // Add b to reg. 1
    STO   r1,temp1      // temp1 = a + b
    CMP   x,temp1       // Test for while condition
    BL    L2            // Continue with loop if x < Temp1
    B     L3            // Terminate loop
L2: LOD   r1,='2'
    MUL   r1,x
    STO   r1,x          // x = 2 * x
    B     L1            // Repeat loop
L3:
```

What are the advantages of a high-level language over machine or assembly language? (1) Machine language (and even assembly language) is difficult to work with and difficult to maintain. (2) With a high-level language you have a much greater degree of machine independence and portability from one kind of computer to another (as long as the other machine has a compiler for that language). (3) You do not have to retrain application programmers every time a new machine (with a new instruction set) is introduced. (4) High-level languages may support data abstraction (through data structures) and program abstraction (procedures and functions).

What are the disadvantages of high-level languages? (1) The programmer does not have complete control of the machine's resources (registers, interrupts, I/O buffers). (2) The compiler may generate inefficient machine language programs. (3) Additional software, the compiler, is needed in order to use a high-level language. As compiler development and hardware have improved over the years, these disadvantages have become less problematic. Consequently, most programming today is done with high-level languages.

An interpreter is software which serves a purpose very similar to that of a compiler. The input to an interpreter is a program written in a high-level

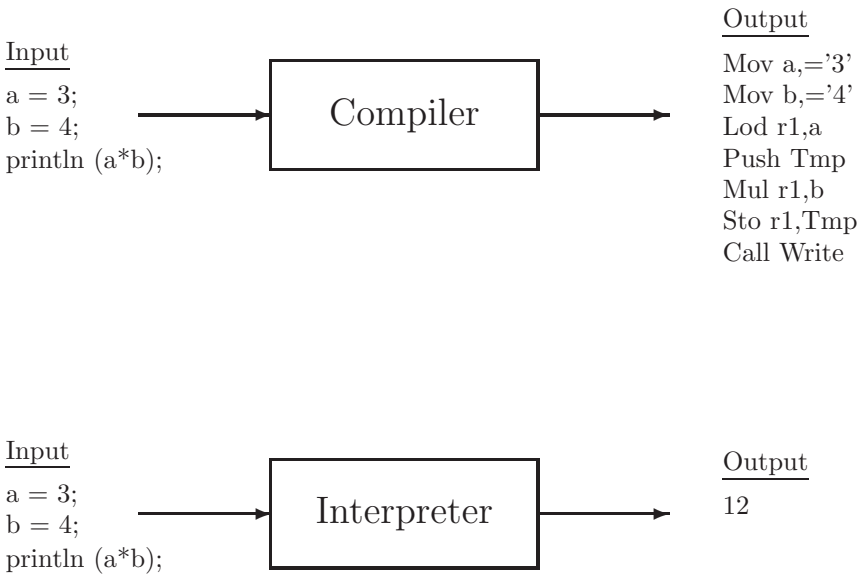


Figure 1.1: A Compiler and Interpreter produce very different output for the same input

language, but rather than generating a machine language program, the interpreter actually carries out the computations specified in the source program. In other words, the output of a compiler is a program, whereas the output of an interpreter is the source program's output. Figure 1.1 shows that although the input may be identical, compilers and interpreters produce very different output. Nevertheless, many of the techniques used in designing compilers are also applicable to interpreters.

Sample Problem 1.1.2

Show the compiler output and the interpreter output for the following Java source code:

```
for (i=1; i<=4; i++) System.out.println (i*3);
```

Solution:

Compiler Output

```
LOD   r1,='4'
```

```
        STO    r1,Temp1
        MOV    i,='1'
L1:     CMP    i,temp1
        BH     L2                  // Terminate loop if i>Temp1
        LOD    r1,i
        MUL    r1,='3'
        STO    r1,Temp2
        PUSH   Temp2               // Parameter for println
        CALL   Print               // Print the result
        B      L1                  // Repeat loop
L2:
```

Interpreter Output

3 6 9 12

Students are often confused about the difference between a compiler and an interpreter. Many commercial compilers come packaged with a built-in edit-compile-run front end. In effect, the student is not aware that after compilation is finished, the object program must be loaded into memory and executed, because this all happens automatically. As larger programs are needed to solve more complex problems, programs are divided into manageable source modules, each of which is compiled separately to an object module. The object modules can then be linked to form a single, complete, machine language program. In this mode, it is more clear that there is a distinction between compile time, the time at which a source program is compiled, and run time, the time at which the resulting object program is loaded and executed. Syntax errors are reported by the compiler at compile time and are shown at the left, below, as compile-time errors. Other kinds of errors not generally detected by the compiler are called run-time errors and are shown at the right below:

Compile-Time Errors	Run-Time Errors
a = ((b+c)*d;	x = a-a;
	y = 100/x; // division by 0
if x<b fn1();	Integer n[ ] = new Integer[7];
else fn2();	n[8] = 16; // invalid subscript

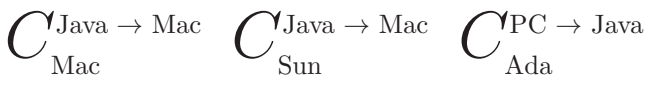


Figure 1.2: Big C notation for compilers. (a) A Java compiler for the Mac. (b) A compiler which translates Java programs to Mac machine language, and which runs on a Sun machine. (c) A compiler which translates PC machine language programs to Java, writen in Ada

It is important to remember that a compiler is a program, and it must be written in some language (machine, assembly, high-level). In describing this program, we are dealing with three languages: (1) the source language, i.e. the input to the compiler, (2) the object language, i.e. the output of the compiler, and (3) the language in which the compiler is written, or the language in which it exists, since it might have been translated into a language foreign to the one in which it was originally written. For example, it is possible to have a compiler that translates Java programs into Macintosh machine language. That compiler could have been written in the C language, and translated into Macintosh (or some other) machine language. Note that if the language in which the compiler is written is a machine language, it need not be the same as the object language. For example, a compiler that produces Macintosh machine language could run on a Sun computer. Also, the object language need not be a machine or assembly language, but could be a high-level language. A concise notation describing compilers is given by Aho et. al. [1] and is shown in Figure 1.2. In these diagrams, the large C stands for Compiler (not the C programming language), the superscript describes the intended translation of the compiler, and the subscript shows the language in which the compiler exists Figure 1.2 (a) shows a Java compiler for the Macintosh. Figure 1.2 (b) shows a compiler which translates Java programs into equivalent Macintosh machine language, but it exists in Sun machine language, and consequently it will run only on a Sun. Figure 1.2 (c) shows a compiler which translates PC machine language programs into equivalent Java programs. It is written in Ada and will not run in that form on any machine.

In this notation the name of a machine represents the machine language for that machine; i.e. Sun represents Sun machine language, and PC represents PC machine language (i.e. Intel Pentium).

Sample Problem 1.1.3

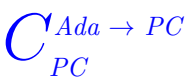
Using the big C notation of Figure 1.2, show each of the following compilers:

1. *An Ada compiler which runs on the PC and compiles to the PC machine language.*

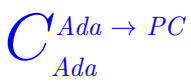
2. *An Ada compiler which compiles to the PC machine language, but which is written in Ada.*
3. *An Ada compiler which compiles to th PC machine language, but which runs on a Sun.*

Solution:

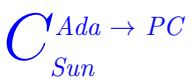
(1)



(2)



(3)



1.1.1 Exercises

1. Show assembly language for a machine of your choice, corresponding to each of the following Java statements:

```
(a)   a = b + c;
(b)   a = (b+c) * (c-d);
(c)   for (i=1; i<=10; i++) a = a+i;
```

2. Show the difference between compiler output and interpreter output for each of the following source inputs:

```
(a)    a = 12;
       b = 6;
       c = a+b;
       println (c+a+b);
```

```
(b)    a = 12;
       b = 6;
       if (a<b) println (a);
       else println (b);
```

```
(c)    a = 12;
       b = 6;
       while (b<a)
       {  a = a-1;
          println (a+b);
       }
```

3. Which of the following Java source errors would be detected at compile time, and which would be detected at run time?

```
(a)     a = b+c = 3;

(b)     if (x<3) a = 2
           else a = x;

(c)     if (a>0) x = 20;
        else if (a<0) x = 10;
             else x = x/a;

(d)     MyClass x [] = new MyClass[100];
        x[100] = new MyClass();
```

4. Using the big C notation, show the symbol for each of the following:

   (a) A compiler which translates COBOL source programs to PC machine language and runs on a PC.
   (b) A compiler, written in Java, which translates FORTRAN source programs to Mac machine language.
   (c) A compiler, written in Java, which translates Sun machine language programs to Java.

1.2 The Phases of a Compiler

The student is reminded that the input to a compiler is simply a string of characters. Students often assume that a particular interpretation is automatically understood by the computer (sum = sum + 1; is obviously an assignment statement, but the computer must be programmed to determine that this is the case). In order to simplify the compiler design and construction process, the compiler is implemented in phases. In general, a compiler consists of at least three phases: (1) lexical analysis, (2) syntax analysis, and (3) code generation. In addition, there could be other optimization phases employed to produce efficient object programs.

1.2.1 Lexical Analysis (Scanner) - Finding the Word Boundaries

The first phase of a compiler is called lexical analysis (and is also known as a lexical scanner). As implied by its name, lexical analysis attempts to isolate the words in an input string. We use the word *word* in a technical sense. A word, also known as a lexeme, a lexical item, or a lexical token, is a string of input characters which is taken as a unit and passed on to the next phase of compilation. Examples of words are:

- key words - while, void, if, for, ...
- identifiers - declared by the programmer
- operators - +, -, *, /, =, ==, ...
- numeric constants - numbers such as 124, 12.35, 0.09E-23, etc.
- character constants - single characters or strings of characters enclosed in quotes
- special characters - characters used as delimiters such as . ( ) , ; :
- comments - ignored by subsequent phases. These must be identified by the scanner, but are not included in the output.

The output of the lexical phase is a stream of tokens corresponding to the words described above. In addition, this phase builds tables which are used by subsequent phases of the compiler. One such table, called the symbol table, stores all identifiers used in the source program, including relevant information and attributes of the identifiers. In block-structured languages it may be preferable to construct the symbol table during the syntax analysis phase because program blocks (and identifier scopes) may be nested. Alternatively, an additional phase called the semantic phase may be used for this purpose.

Sample Problem 1.2.1

Show the token classes, or "words", put out by the lexical analysis phase corresponding to this Java source input:

```
sum = sum + unit * /* accumulate sum */ 1.2e-12 ;
```

Solution:

identifier	*(sum)*
assignment	*(=)*
identifier	*(sum)*
operator	*(+)*
identifier	*(unit)*
operator	*(*)*
numeric constant	*(1.2e-12)*
semicolon	*(;)*

1.2.2 Syntax Analysis Phase

The syntax analysis phase is often called the *parser*. This term is critical to understanding both this phase and the study of languages in general. The parser will check for proper syntax, issue appropriate error messages, and determine the underlying structure of the source program. The output of this phase may be a stream of atoms or a collection of syntax trees. An atom is an atomic operation, or one that is generally available with one (or just a few) machine language instruction(s) on most target machines. For example, MULT, ADD, and MOVE could represent atomic operations for multiplication, addition, and moving data in memory. Each operation could have 0 or more operands also listed in the atom: (operation, operand1, operand2, operand3). The meaning of the following atom would be to add A and B, and store the result into C:

(ADD, A, B, C)

In Sample Problem 1.2.2, below, each atom consists of three or four parts: an operation, one or two operands, and a result. Note that the compiler must put out the MULT atom before the ADD atom, despite the fact that the addition is encountered first in the source statement.

Sample Problem 1.2.2

Show the atoms corresponding to the following Java statement:

```
a = b + c * d ;
```

Solution:

```
(MULT, c, d, temp1)
(ADD, b, temp1, temp2)
(MOVE, temp2, a)
```

To implement transfer of control, we could use label atoms, which serve only to mark a spot in the object program to which we might wish to branch in implementing a control structure such as if or while. A label atom with the name L1 would be (LBL,L1). We could use a jump atom for an unconditional branch, and a test atom for a conditional branch: The atom (JMP, L1) would be an unconditional branch to the label L1. The atom (TEST, a, ¡=, b, L2) would be a conditional branch to the label L2, if a¡=b is true.

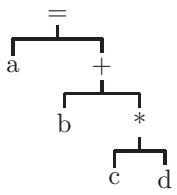


Figure 1.3: A Syntax Tree for a = b + c * d

Sample Problem 1.2.3

Show the atoms corresponding to the following Java statement:

```
while (a <= b) a = a + 1;
```

Solution:

```
(LBL, L1)
(Test, a, <=, b, L2)
(JMP, L3)
(LBL, L2)
(ADD, a, 1, a)
(JMP, L1)
(LBL, L3)
```

Some parsers put out syntax trees as an intermediate data structure, rather than atom strings. A syntax tree indicates the structure of the source statement, and object code can be generated directly from the syntax tree. A syntax tree for the expression $a = b + c * d$ is shown in Figure 1.3.

In syntax trees, each interior node represents an operation or control structure and each leaf node represents an operand. A statement such as if (Expr) Stmt1 else Stmt2 could be implemented as a node having three children: one for the conditional expression, one for the true part (Stmt1), and one for the else statement (Stmt2). The while control structure would have two children: one for the loop condition, and one for the statement to be repeated. The compound statement could be treated a few different ways. The compound statement could have an unlimited number of children, one for each statement in the compound statement. The other way would be to treat the semicolon like a statement concatenation operator, yielding a binary tree.

Once a syntax tree has been created, it is not difficult to generate code from the syntax tree; a postfix traversal of the tree is all that is needed. In a postfix traversal, for each node, N, the algorithm visits all the subtrees of N, and visits the node N last, at which point the instruction(s) corresponding to node N can be generated.

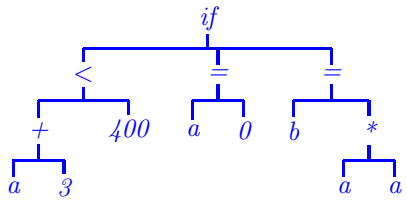
Sample Problem 1.2.4

Show a syntax tree for the Java statement:

```
if (a+3 < 400) a =0; else b = a*a;
```

Assume that an `if` *statement consists of three subtrees, one for the condition, one for the consequent statement, and one for the* `else` *statement, if necessary.*

Solution:



Many compilers also include a phase for semantic analysis. In this phase the data types are checked, and type conversions are performed when necessary. The compiler may also be able to detect some semantic errors, such as division by zero, or the use of a null pointer.

1.2.3 Global Optimization

The global optimization phase is optional. Its purpose is simply to make the object program more efficient in space and/or time. It involves examining the sequence of atoms put out by the parser to find redundant or unnecessary instructions or inefficient code. Since it is invoked before the code generator, this phase is often called machine-independent optimization. For example, in the following program segment:

```
stmt1
go to label1
stmt2
stmt3
label2: stmt4
```

stmt2 and stmt3 can never be executed. They are unreachable and can be eliminated from the object program. A second example of global optimization is shown below:

```
for (i=1; i<=100000; i++)
{  x = Math.sqrt (y);  // square root method
System.out.println (x+i);
}
```

In this case, the assignment to x need not be inside the loop since y does not change as the loop repeats (it is a loop invariant). In the global optimization phase, the compiler would move the assignment to x out of the loop in the object program:

```
x = Math.sqrt (y); // loop invariant
for (i=1; i<=100000; i++)
System.out.println (x+i);
```

This would eliminate 99,999 unnecessary calls to the sqrt method at run time.

The reader is cautioned that global optimization can have a serious impact on run-time debugging. For example, if the value of y in the above example was negative, causing a run-time error in the sqrt function, the user would be unaware of the actual location of that portion of code which called the sqrt function, because the compiler would have moved the offending statement (usually without informing the programmer). Most compilers that perform global optimization also have a switch with which the user can turn optimization on or off. When debugging the program, the switch would be off. When the program is correct, the switch would be turned on to generate an optimized version for the user. One of the most difficult problems for the compiler writer is making sure that the compiler generates optimized and unoptimized object modules, from the same source module, which are equivalent.

1.2.4 Code Generation

Most Java compilers produce an intermediate form, known as *byte code*, which can be interpreted by the Java run-time environment. In this book we will be assuming that our compiler is to produce native code for a particular machine.

It is assumed that the student has had some experience with assembly language and machine language, and is aware that the computer is capable of executing only a limited number of primitive operations on operands with numeric

memory addresses, all encoded as binary values. In the code generation phase, atoms or syntax trees are translated to machine language (binary) instructions, or to assembly language, in which case the assembler is invoked to produce the object program. Symbolic addresses (statement labels) are translated to relocatable memory addresses at this time.

For target machines with several CPU registers, the code generator is responsible for register allocation. This means that the compiler must be aware of which registers are being used for particular purposes in the generated program, and which become available as code is generated.

For example, an ADD atom might be translated to three machine language instructions: (1) load the first operand into a register, (2) add the second operand to that register, and (3) store the result, as shown for the atom (ADD, a, b,temp):

```
LOD r1,a // Load a into reg. 1
ADD r1,b // Add b to reg. 1
STO r1,temp // Store reg. 1 in temp
```

In Sample Problem 1.2.5 the destination for the MOV instruction is the first operand, and the source is the second operand, which is the reverse of the operand positions in the MOVE atom.

Sample Problem 1.2.5

Show assembly language instructions corresponding to the following atom string:

```
(ADD, a, b, temp1)
(TEST, a, ==, b, L1)
(MOVE, temp1, a)
(LBL, L1)
(MOVE, temp1, b)
```

Solution:

```
        LOD   r1,a
        ADD   r1,b
        STO   r1,temp1     // ADD, a, b, temp1
        CMP   a,b
        BE    L1           // TEST, A, ==, B, L1
        MOV   a,temp1      // MOVE, temp1, a
L1:     MOV   b,temp1      // MOVE, temp1, b
```

It is not uncommon for the object language to be another high-level language. This is done in order to improve portablility of the language being implemented.

1.2.5 Local Optimization

The local optimization phase is also optional and is needed only to make the object program more efficient. It involves examining sequences of instructions put out by the code generator to find unnecessary or redundant instructions. For this reason, local optimization is often called machine-dependent optimization. An example of a local optimization would be a load/store optimization. An addition operation in the source program might result in three instructions in the object program: (1) Load one operand into a register, (2) add the other operand to the register, and (3) store the result. Consequently, the expression `a + b + c` in the source program might result in the following instructions as code generator output:

```
LOD r1,a            // Load a into register 1
ADD  r1,b            // Add b to register 1
STO r1,temp1        // Store the result in temp1*
LOD r1,temp1        // Load result into reg 1*
ADD r1,c            // Add c to register 1
STO r1,temp2        // Store the result in temp2
```

Note that some of these instructions (those marked with * in the comment) can be eliminated without changing the effect of the program, making the object program both smaller and faster:

```
LOD r1,a            // Load a into register 1
ADD r1,b            // Add b to register 1
ADD r1,c            // Add c to register 1
STO r1,temp         // Store the result in temp
```

A diagram showing the phases of compilation and the output of each phase is shown in Figure 1.4. Note that the optimization phases may be omitted (i.e. the atoms may be passed directly from the Syntax phase to the Code Generator, and the instructions may be passed directly from the Code Generator to the compiler output file.)

A word needs to be said about the flow of control between phases. One way to handle this is for each phase to run from start to finish separately, writing output to a disk file. For example, lexical analysis is started and creates a file of tokens. Then, after the entire source program has been scanned, the syntax

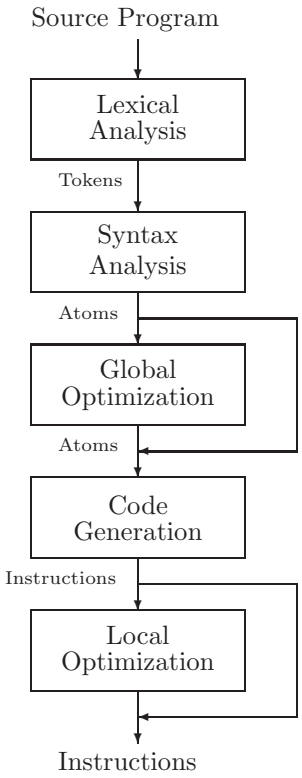


Figure 1.4: The Phases of a Compiler

analysis phase is started, reads the entire file of tokens, and creates a file of atoms. The other phases continue in this manner; this would be a multiple pass compiler since the input is scanned several times.

Another way for flow of control to proceed would be to start up the syntax analysis phase first. Each time it needs a token it calls the lexical analysis phase as a subroutine, which reads enough source characters to produce one token, and returns it to the parser. Whenever the parser has scanned enough source code to produce an atom, the atom is converted to object code by calling the code generator as a subroutine; this would be a single pass compiler.

1.2.6 Exercises

1. Show the lexical tokens corresponding to each of the following Java source inputs:

```
(a)    for (i=1; i<5.1e3; i++) func1(x);
(b)    if (sum!=133) /* sum = 133 */
(c)    while ( 1.3e-2 if &&
(d)    if 1.2.3 < 6
```

2. Show the sequence of atoms put out by the parser, and show the syntax tree corresponding to each of the following Java source inputs:

```
(a)    a = (b+c) * d;
(b)    if (a<b) a = a + 1;
(c)    while (x>1)
       { x = x/2;
         i = i+1;
       }
(d)    a = b - c - d/a + d * a;
```

3. Show an example of a Java statement which indicates that the order in which the two operands of an ADD are evaluated can cause different results:

```
operand1 + operand2
```

4. Show how each of the following Java source inputs can be optimized using global optimization techniques:

```
(a)     for (i=1; i<=10; i++)
        {   x = i + x;
            a[i] = a[i-1];
            y = b * 4;
```

```
        }

(b)     for (i=1; i<=10; i++)
        {   x = i;
            y = x/2;
            a[i] = x;
        }

(c)     if (x>0) {x = 2; y = 3;}
        else {y = 4; x = 2;}


(d)     if (x>0) x = 2;
        else if (x<=0) x = 3;
             else x = 4;
```

5. Show, in assembly language for a machine of your choice, the output of the code generator for the following atom string:

```
(ADD,a,b,temp1)
(SUB,c,d,temp2)
(TEST,temp1,<,temp2,L1)
(JUMP,L2)
(LBL,L1)
(MOVE,a,b)
(JUMP,L3)
(LBL,L2)
(MOVE,b,a)
(LBL,L3)
```

6. Show a Java source statement which might have produced the atom string in Problem 5, above.

7. Show how each of the following object code segments could be optimized using local optimization techniques:

```
(a)       LD     r1,a
          MULT   r1,b
          ST     r1,temp1
          LD     r1,temp1
          ADD    r1,C
          ST     r1,temp2

(b)       LD     r1,a
          ADD    r1,b
          ST     r1,temp1
```

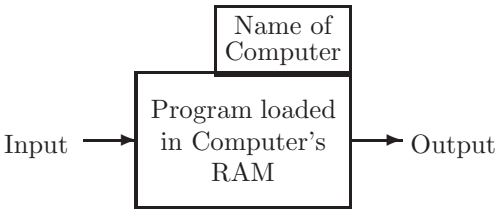


Figure 1.5: Notation for a program running on a computer

```
            MOV    c,temp1

(c)         CMP     a,b
            BH      L1
            B       L2
     L1:    MOV     a,b
            B       L3
     L2:    MOV     b,a
L3:
```

1.3 Implementation Techniques

By this point it should be clear that a compiler is not a trivial program. A new compiler, with all optimizations, could take over a person-year to implement. For this reason, we are always looking for techniques or shortcuts which will speed up the development process. This often involves making use of compilers, or portions of compilers, which have been developed previously. It also may involve special compiler generating tools, such as lex and yacc , which are part of the Unix environment, or newer tools such as JavaCC or SableCC.

In order to describe these implementation techniques graphically, we use the method shown in Figure 1.5, in which the computer is designated with a rectangle, and its name is in a smaller rectangle sitting on top of the computer. In all of our examples the program loaded into the computer's memory will be a compiler. It is important to remember that a computer is capable of running only programs written in the machine language of that computer. The input and output (also compilers in our examples) to the program in the computer are shown to the left and right, respectively.

Since a compiler does not change the purpose of the source program, the superscript on the output is the same as the superscript on the input ($X \rightarrow Y$), as shown in Figure 1.6. The subscript language (the language in which it exists) of the executing compiler (the one inside the computer), M, must be the machine language of the computer on which it is running. The subscript language of the input, S, must be the same as the source language of the executing compiler. The subscript language of the output, O, must be the same as the object language of the executing compiler.

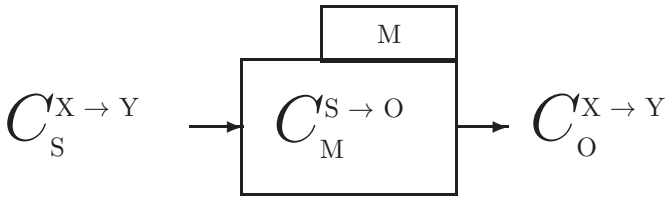
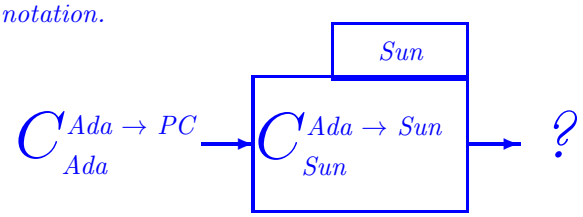


Figure 1.6: Notation for a compiler being translated to a different language

In the following sections it is important to remember that a compiler does not change the purpose of the source program; a compiler translates the source program into an equivalent program in another language (the object program). The source program could, itself, be a compiler. If the source program is a compiler which translates language A into language B, then the object program will also be a compiler which translates language A into language B.

Sample Problem 1.3.1

Show the output of the following compilation using the big C notation.



Solution:

$$C^{Ada \to PC}_{Sun}$$

1.3.1 Bootstrapping

The term bootstrapping is derived from the phrase "pull yourself up by your bootstraps" and generally involves the use of a program as input to itself (the student may be familiar with bootstrapping loaders which are used to initialize

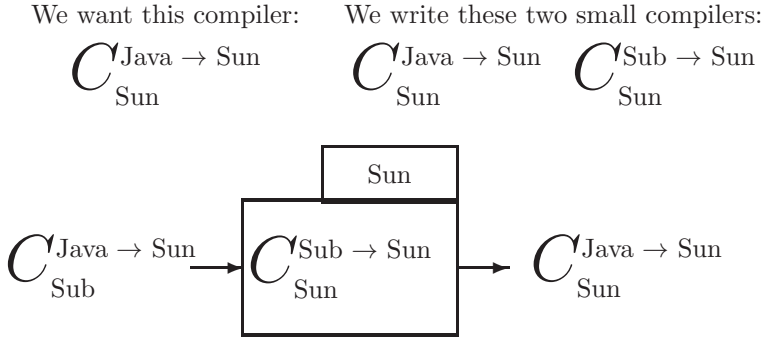


Figure 1.7: Bootstrapping Java onto a Sun computer

a computer just after it has been switched on, hence the expression "to boot" a computer).

In this case, we are talking about bootstrapping a compiler, as shown in Figure 1.7. We wish to implement a Java compiler for the Sun computer. Rather than writing the whole thing in machine (or assembly) language, we instead choose to write two easier programs. The first is a compiler for a subset of Java, written in machine (assembly) language. The second is a compiler for the full Java language written in the Java subset language. In Figure 1.7 the subset language of Java is designated 'Sub', and it is simply Java, without several of the superfluous features, such as enumerated types, unions, switch statements, etc. The first compiler is loaded into the computer's memory and the second is used as input. The output is the compiler we want i.e. a compiler for the full Java language, which runs on a Sun and produces object code in Sun machine language.

In actual practice this is an iterative process, beginning with a small subset of Java, and producing, as output, a slightly larger subset. This is repeated, using larger and larger subsets, until we eventually have a compiler for the complete Java language.

1.3.2 Cross Compiling

New computers with enhanced (and sometimes reduced) instruction sets are constantly being produced in the computer industry. The developers face the problem of producing a new compiler for each existing programming language each time a new computer is designed. This problem is simplified by a process called cross compiling.

Cross compiling is a two-step process and is shown in Figure 1.8. Suppose that we have a Java compiler for the Sun, and we develop a new machine called a Mac. We now wish to produce a Java compiler for the Mac without writing it entirely in machine (assembly) language; instead, we write the compiler in Java. Step one is to use this compiler as input to the Java compiler on the Sun. The output is a compiler that translates Java into Mac machine language, and

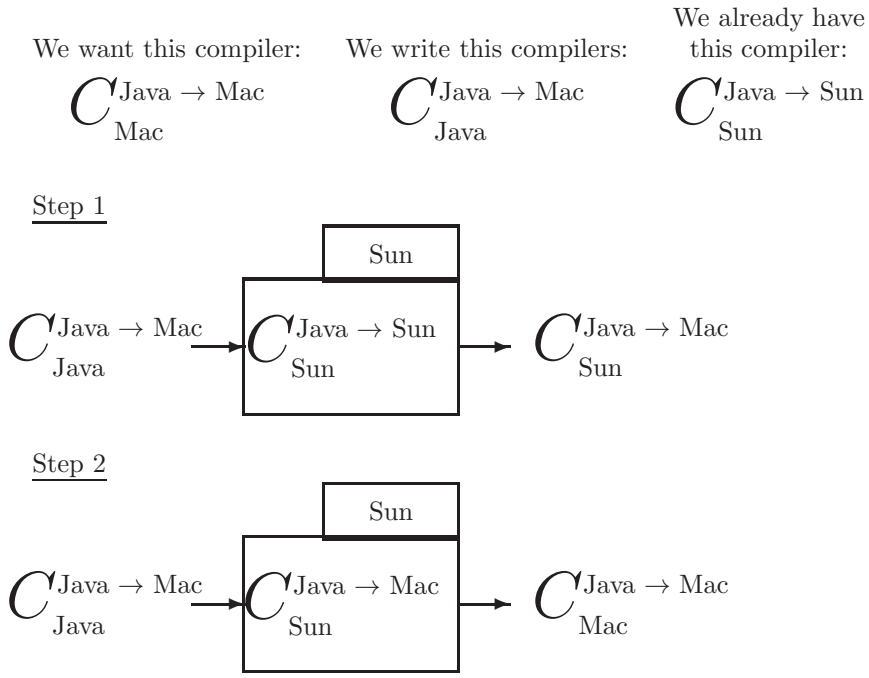


Figure 1.8: Cross compiling Java from a Sun to a Mac computer

which runs on a Sun. Step two is to load this compiler into the Sun and use the compiler we wrote in Java as input once again. This time the output is a Java compiler for the Mac which runs on the Mac, i.e. the compiler we wanted to produce.

Note that this entire process can be completed before a single Mac has been built. All we need to know is the architecture (the instruction set, instruction formats, addressing modes, ...) of the Mac.

1.3.3 Compiling To Intermediate Form

As we mentioned in our discussion of interpreters above, it is possible to compile to an intermediate form, which is a language somewhere between the source high-level language and machine language. The stream of atoms put out by the parser is a possible example of an intermediate form. The primary advantage of this method is that one needs only one translator for each high-level language to the intermediate form (each of these is called a front end) and only one translator (or interpreter) for the intermediate form on each computer (each of these is called a back end). As depicted in Figure 1.9, for three high-level languages and two computers we would need three translators to intermediate form and two code generators (or interpreters), one for each computer. Had we not used the intermediate form, we would have needed a total of six different

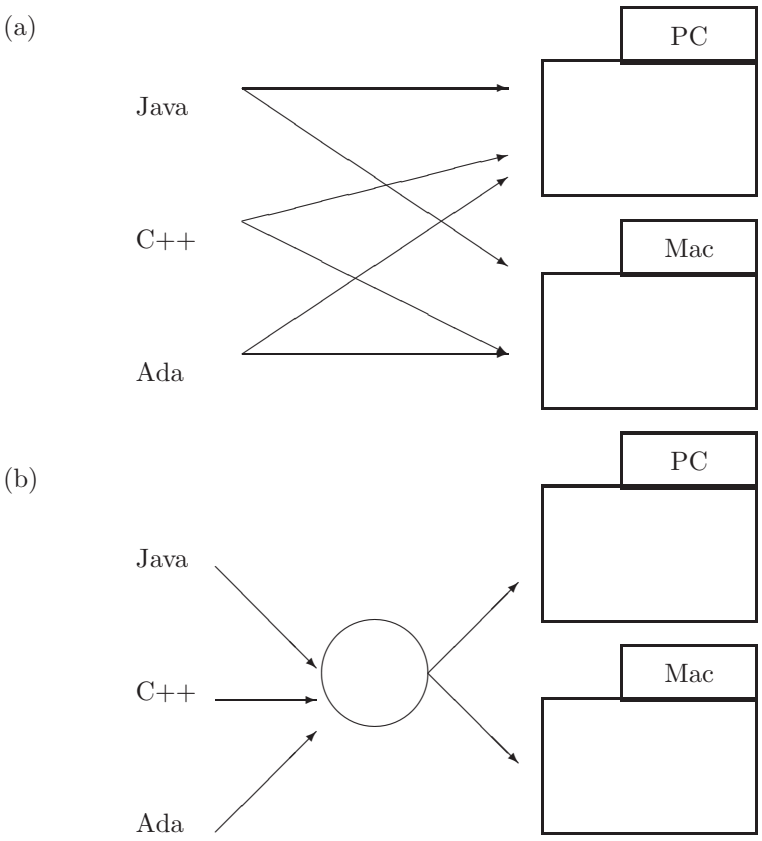


Figure 1.9: (a) Six compilers neede for three languages on two machines. (b) Fewer than three compilers using intermediate form needed for the same languages and machines.

compilers. In general, given n high-level languages and m computers, we would need n x m compilers. Assuming that each front end and each back end is half of a compiler, we would need (n+m)/2 compilers using intermediate form.

A very popular intermediate form for the PDP-8 and Apple II series of computers, among others, called p-code, was developed several years ago at the University of California at San Diego. Today, high-level languages such as C are commonly used as an intermediate form. The Java Virtual Machine (i.e. Java byte code) is another intermediate form which has been used extensively on the Internet.

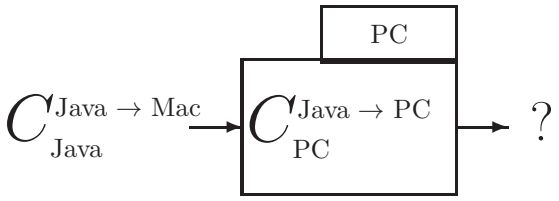
1.3.4 Compiler-Compilers

Much of compiler design is understood so well at this time that the process can be automated. It is possible for the compiler writer to write specifications of the source language and of the target machine so that the compiler can be generated automatically. This is done by a compiler-compiler. We will introduce this topic in Chapters 2 and 5 when we study the SableCC public domain software.

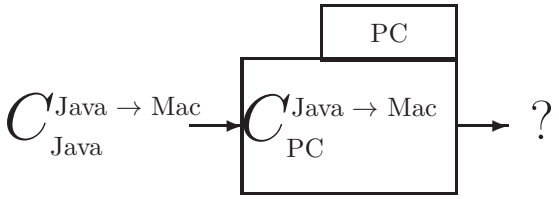
1.3.5 Exercises

1. Fill in the missing information in the compilations indicated below:

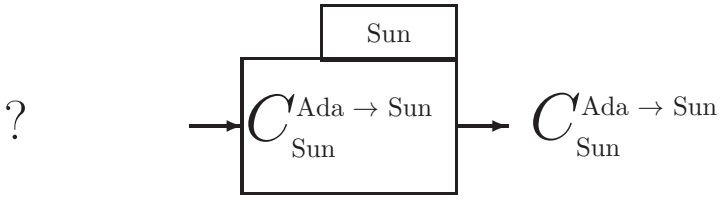
(a)



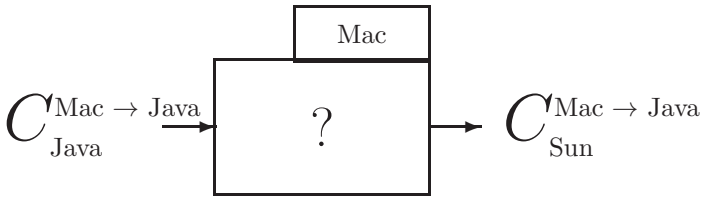
(b)



(c)



(d)



2. How could the compiler generated in part (d) of the previous question be used?

3. If the only computer you have is a PC (for which you already have a FORTRAN compiler), show how you can produce a FORTRAN compiler for the Mac computer, without writing any assembly or machine language.

4. Show how Ada can be bootstrapped in two steps on a Sun. First use a small subset of Ada, *Sub1* to build a compiler for a larger subset, *Sub2* (by bootstrapping). Then use *Sub2* to implement Ada (again by bootstrapping). *Sub1* is a subset of *Sub2*.

5. You have 3 computers: A PC, a Mac, and a Sun. Show how to generate automatically a java to FORT translator which will run on a Sun if you also have the four compilers shown below:
$C^{\text{Java} \to \text{FORT}}_{\text{Mac}}$ $C^{\text{FORT} \to \text{Java}}_{\text{Sun}}$ $C^{\text{Java} \to \text{Sun}}_{\text{Mac}}$ $C^{\text{Java} \to \text{FORT}}_{\text{Java}}$

6. In Figure 1.8, suppose we also have $C^{\text{Java} \to \text{Sun}}_{\text{Java}}$. When we write $C^{\text{Java} \to \text{Mac}}_{\text{Java}}$, which of the phases of $C^{\text{Java} \to \text{Sun}}_{\text{Java}}$ can be reused as is?

7. Using the big C notation, show the 11 translators which are represented in Figure 1.9. Use *Int* to represent the intermediate form.

1.4 Case Study: Decaf

As we study the various phases of compilation and techniques used to implement those phases, we will show how the concepts can be applied to an actual compiler. For this purpose we will define a language called Decaf as a relatively simple subset of the Java language. The implementation of Decaf will then be used as a case study, or extended project, throughout the textbook. The last section of each chapter will show how some of the concepts of that chapter can be used in the design of an actual compiler for Decaf.

Decaf is a "bare bones" version of Java. Its only data types are int and float, and it does not permit arrays, classes, enumerated types, methods, or subprograms. However, it does include while, for, and if control structures, and it is possible to write some useful programs in Decaf. The example that we will

use for the case study is the following Decaf program, to compute the cosine function:

```
class Cosine
{ public static void main (String [] args)
{ float cos, x, n, term, eps, alt;
/* compute the cosine of x to within tolerance eps */
/* use an alternating series */

x = 3.14159;
eps = 0.0001;
n = 1;
cos = 1;
term = 1;
alt = -1;
while (term>eps

{ term = term * x * x / n / (n+1);
  cos = cos + alt * term;
  alt = -alt;
  n = n + 2;
}
}
}
```

This program computes the cosine of the value x (in radians) using an alternating series which terminates when a term becomes smaller than a given tolerance (eps). This series is described in most calculus textbooks and can be written as:

$cos(x) = 1 - x^2/2 + x^4/24 - x^6/720 + ...$

Note that in the statement $term = term * x * x/n/(n+1)$ the multiplication and division operations associate to the left, so that n and $(n+1)$ are both in the denominator.

A precise specification of Decaf, similar to a BNF description, is given in Appendix A. The lexical specifications (free format, white space taken as delimiters, numeric constants, comments, etc.) of Decaf are the same as standard C.

When we discuss the back end of the compiler (code generation and optimization) we will need to be concerned with a target machine for which the compiler generates instructions. Rather than using an actual computer as the target machine, we have designed a fictitious computer called Mini as the target machine. This was done for two reasons: (1) We can simplify the architecture of the machine so that the compiler is not unnecessarily complicated by complex addressing modes, complex instruction formats, operating system constraints, etc., and (2) we provide the source code for a simulator for Mini so that the student can compile and execute Mini programs (as long as he/she has a C

compiler on his/her computer). The student will be able to follow all the steps in the compilation of the above cosine program, understand its implementation in Mini machine language, and observe its execution on the Mini machine.

The complete source code for the Decaf compiler and the Mini simulator is provided in the appendix and is available through the Internet, as described in the appendix. With this software, the student will be able to make his/her own modifications to the Decaf language, the compiler, or the Mini machine architecture. Some of the exercises in later chapters are designed with this intent.

1.5 Chapter Summary

This chapter reviewed the concepts of high-level language and machine language and introduced the purpose of the compiler. The compiler serves as a translator from any program in a given high-level language (the source program) to an equivalent program in a given machine language (the object program). We stressed the fact that the output of a compiler is a program, and contrasted compilers with interpreters, which carry out the computations specified by the source program.

We introduced the phases of a compiler: (1) The lexical scanner finds word boundaries and produces a token corresponding to each word in the source program. (2) The syntax phase, or parser, checks for proper syntax and, if correct, puts out a stream of atoms or syntax trees which are similar to the primitive operations found in a typical target machine. (3) The global optimization phase is optional, eliminates unnecessary atoms or syntax tree elements, and improves efficiency of loops if possible. (4) The code generator converts the atoms or syntax trees to instructions for the target machine. (5) The local optimization phase is also optional, eliminates unnecessary instructions, and uses other techniques to improve the efficiency of the object program.

We discussed some compiler implementation techniques. The first implementation technique was bootstrapping, in which a small subset of the source language is implemented and used to compile a compiler for the full source language, written in the source language itself. We also discussed cross compiling, in which an existing compiler can be used to implement a compiler for a new computer. We showed how the use of an intermediate form can reduce the workload of the compiler writer.

Finally, we examined a language called Decaf, a small subset of Java, which will be used for a case study compiler throughout the textbook.

Chapter 2

Lexical Analysis

In this chapter we study the implementation of lexical analysis for compilers. As defined in Chapter 1, lexical analysis is the identification of words in the source program. These words are then passed as tokens to subsequent phases of the compiler, with each token consisting of a class and value. The lexical analysis phase can also begin the construction of tables to be used later in the compilation; a table of identifiers (symbol table) and a table of numeric constants are two examples of tables which can be constructed in this phase of compilation.

However, before getting into lexical analysis we need to be sure that the student understands those concepts of formal language and automata theory which are critical to the design of the lexical analyser. The student who is familiar with regular expressions and finite automata may wish to skip or skim section 2.0 and move on to lexical analysis in section 2.2.1.

2.0 Formal Languages

This section introduces the subject of formal languages, which is critical to the study of programming languages and compilers. A formal language is one that can be specified precisely and is amenable for use with computers, whereas a natural language is one which is normally spoken by people. The syntax of Java is an example of a formal language, but it is also possible for a formal language to have no apparent meaning or purpose, as discussed in the following sections.

2.0.1 Language Elements

Before we can define a language, we need to make sure the student understands some fundamental definitions from discrete mathematics. A set is a collection of unique objects. In listing the elements of a set, we normally list each element only once (though it is not incorrect to list an element more than once), and the elements may be listed in any order. For example, {boy, girl, animal} is a

set of words, but it represents the same set as {girl, boy, animal, girl}. A set may contain an infinite number of objects. The set which contains no elements is still a set, and we call it the empty set and designate it either by { } or by ϕ.

A string is a list of characters from a given alphabet. The elements of a string need not be unique, and the order in which they are listed is important. For example, "abc" and "cba" are different strings, as are "abb" and "ab". The string which consists of no characters is still a string (of characters from the given alphabet), and we call it the null string and designate it by ϵ. It is important to remember that if, for example, we are speaking of strings of zeros and ones (i.e. strings from the alphabet {0,1}), then ϵ is a string of zeros and ones.

In this and following chapters, we will be discussing languages. A (formal) language is a set of strings from a given alphabet. In order to understand this, it is critical that the student understand the difference between a set and a string and, in particular, the difference between the empty set and the null string. The following are examples of languages from the alphabet {0,1}:

1. {0,10,1011}
2. { }
3. {ϵ,0,00,000,0000,00000,... }
4. The set of all strings of zeroes and ones having an even number of ones.

The first two examples are finite sets while the last two examples are infinite. The first two examples do not contain the null string, while the last two examples do. The following are four examples of languages from the alphabet of characters available on a computer keyboard:

1. {0,10,1011}
2. {ϵ}
3. Java syntax
4. Italian syntax

The third example is the syntax of a programming language (in which each string in the language is a Java program without syntax errors), and the fourth example is a natural language (in which each string in the language is a grammatically correct Italian sentence). The second example is not the empty set.

2.0.2 Finite State Machines

We now encounter a problem in specifying, precisely, the strings in an infinite (or very large) language. If we describe the language in English, we lack the precision necessary to make it clear exactly which strings are in the language and which are not in the language. One solution to this problem is to use a

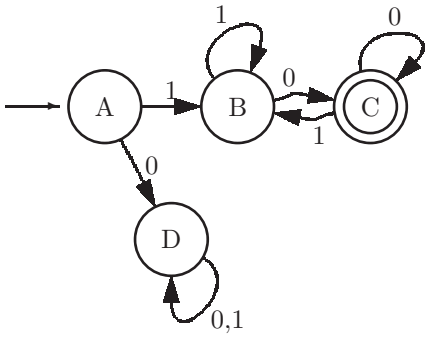


Figure 2.1: Example of a finite state machine

mathematical or hypothetical machine called a finite state machine. This is a machine which we will describe in mathematical terms and whose operation should be perfectly clear, though we will not actually construct such a machine. The study of theoretical machines such as the finite state machine is called automata theory because *automaton* is just another word for *machine*. A finite state machine consists of:

1. A finite set of states, one of which is designated the starting state, and zero or more of which are designated accepting states. The starting state may also be an accepting state.

2. A state transition function which has two arguments: a state and an input symbol (from a given input alphabet). It returns a state as its result.

Here is how the machine works. The input is a string of symbols from the input alphabet. The machine is initially in the starting state. As each symbol is read from the input string, the machine proceeds to a new state as indicated by the transition function, which is a function of the input symbol and the current state of the machine. When the entire input string has been read, the machine is either in an accepting state or in a non-accepting state. If it is in an accepting state, then we say the input string has been accepted. Otherwise the input string has not been accepted, i.e. it has been rejected. The set of all input strings which would be accepted by the machine form a language, and in this way the finite state machine provides a precise specification of a language.

Finite state machines can be represented in many ways, one of which is a state diagram. An example of a finite state machine is shown in Figure 2.1. Each state of the machine is represented by a circle, and the transition function is represented by arcs labeled by input symbols leading from one state to another. The accepting states are double circles, and the starting state is indicated by an arc with no state at its source (tail) end.

For example, in Figure 2.1. if the machine is in state B and the input is a 0, the machine enters state C. If the machine is in state B and the input is a 1, the machine stays in state B. State A is the starting state, and state C is the only accepting state. This machine accepts any string of zeroes and ones which begins with a one and ends with a zero, because these strings (and only

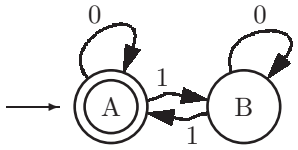


Figure 2.2: Even parity checker

	0	1
A	D	B
B	C	B
*C	C	B
D	D	D

(a)

	0	1
*A	A	B
B	B	A

(b)

Figure 2.3: Finite state machines in table form for the machines of (a) Figure 2.1 and (b) Figure 2.2

these) will cause the machine to be in an accepting state when the entire input string has been read. Another finite state machine is shown in Figure 2.2. This machine accepts any string of zeroes and ones which contains an even number of ones (which includes the null string). Such a machine is called a parity checker. For both of these machines, the input alphabet is {0,1}.

Notice that both of these machines are completely specified, and there are no contradictions in the state transitions. This means that for each state there is exactly one arc leaving that state labeled by each possible input symbol. For this reason, these machines are called deterministic. We will be working only with deterministic finite state machines.

Another representation of the finite state machine is the table, in which we assign names to the states (A, B, C, ...) and these label the rows of the table. The columns are labeled by the input symbols. Each entry in the table shows the next state of the machine for a given input and current state. The machines of Figure 2.1. and Figure 2.2. are shown in table form in Figure 2.3 Accepting states are designated with an asterisk, and the starting state is the first one listed in the table.

With the table representation it is easier to ensure that the machine is completely specified and deterministic (there should be exactly one entry in every cell of the table). However, many students find it easier to work with the state diagram representation when designing or analyzing finite state machines.

Sample Problem 2.0.1

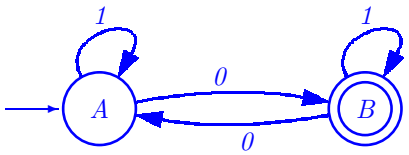
Show a finite state machine, in either state graph or table form,

for each of the following languages (in each case the input alphabet is {0,1}):

1. *Strings containing an odd number of zeros*
2. *Strings containing three consecutive ones*
3. *Strings containing exactly three zeros*
4. *Strings containing an odd number of zeros and an even number of ones.*

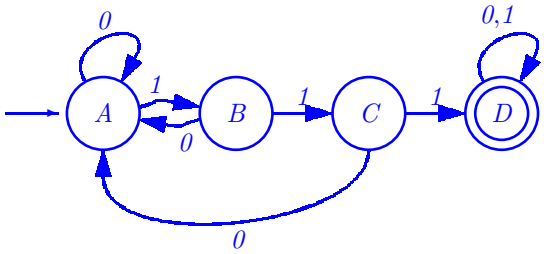
Solution:

Solution for #1



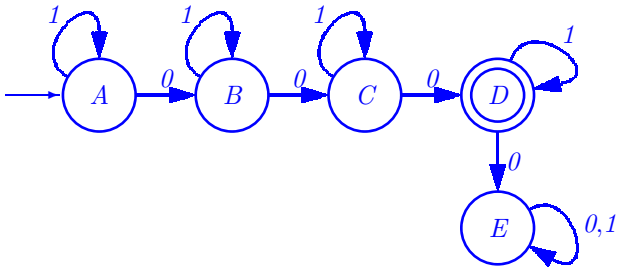
	0	1
A	B	A
*B	A	B

Solution for #2



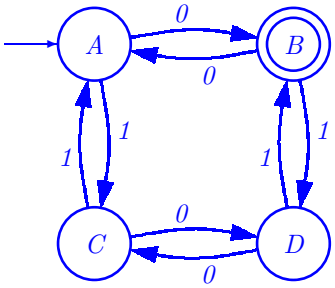
	0	1
A	A	B
B	A	C
C	A	D
*D	D	D

Solution for #3



	0	1
A	B	A
B	C	B
C	D	C
*D	E	D
E	E	E

Solution for #4



	0	1
A	B	C
*B	A	D
C	D	A
D	C	B

2.0.3 Regular Expressions

Another method for specifying languages is regular expressions. These are formulas or expressions consisting of three possible operations on languages: union, concatenation, and Kleene star.

(1) *Union* Since a language is a set, this operation is the union operation as defined in set theory. The union of two sets is that set which contains all the elements in each of the two sets and nothing else. The union operation on languages is designated with a '+'. For example,

```
{abc, ab, ba} + {ba, bb} = {abc, ab, ba, bb}
```

Note that the union of any language with the empty set is that language:

```
L + {} = L
```

(2) *Concatenation* In order to define concatenation of languages, we must first define concatenation of strings. This operation will be designated by a raised dot (whether operating on strings or languages), which may be omitted. This is simply the juxtaposition of two strings forming a new string. For example,

abc · ba = abcba

Note that any string concatenated with the null string is that string itself: $s \cdot \epsilon = s$.

In what follows, we will omit the quote marks around strings to avoid cluttering the page needlessly. The concatenation of two languages is that language formed by concatenating each string in one language with each string in the other language. For example,

$\{ab, a, c\} \cdot \{b, \epsilon\} = \{ab \cdot b, ab \cdot \epsilon, a \cdot b, a \cdot \epsilon, c \cdot b, c \cdot \epsilon\} = \{abb, ab, a, cb, c\}$

In this example, the string ab need not be listed twice. Note that if L1 and L2 are two languages, then L1 · L2 is not necessarily equal to L2 · L1. Also, L $\cdot\{\epsilon\}$ = L, but L $\cdot\phi = \phi$.

(3) *Kleene ** This operation is a unary operation (designated by a postfix asterisk) and is often called closure. If L is a language, we define:

$L^0 = \{\epsilon\}$
$L^1 = L$
$L^2 = L \cdot L$
$\vdots$
$L^n = L \cdot L^{n-1}$
$L* = L^0 + L^1 + L^2 + L^3 + L^4 + L^5 + ...$

Note that $\phi* = \{\epsilon\}$. Intuitively, Kleene * generates zero or more concatenations of strings from the language to which it is applied.

We will use a shorthand notation in regular expressions: if x is a character in the input alphabet, then `x = {'x'}`; i.e., the character x represents the set consisting of one string of length 1 consisting of the character x. This simplifies some of the regular expressions we will write:

$0 + 1 = \{0\} + \{1\} = \{0, 1\}$
$0 + \epsilon = \{0, \epsilon\}$

A regular expression is an expression involving the above three operations and languages. Note that Kleene * is unary (postfix) and the other two operations are binary. Precedence may be specified with parentheses, but if parentheses are omitted, concatenation takes precedence over union, and Kleene * takes precedence over concatenation. If L1 , L2 and L3 are languages, then:

$L1 + L2 \cdot L3 = L1 + (L2 \cdot L3)$
$L1 \cdot L2* = L1 \cdot (L2*)$

An example of a regular expression is: (0+1)* To understand what strings are in this language, let L = {0,1}. We need to find L*: $L^0 = \{\epsilon\}$
$L^1 = \{0, 1\}$
$L^2 = L \cdot L^1 = \{00, 01, 10, 11\}$
$L^3 = L \cdot L^2 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
$\vdots$
$L* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, ...\}$
This is the set of all strings of zeros and ones.

Another example: $(0+1)*.0 = 1(0+1)*0 = \{10, 100, 110, 1000, 1010, 1100, 1110, ...\}$ This is the set of all strings of zeros and ones which begin with a 1 and end with a 0.

Note that we do not need to be concerned with the order of evaluation of several concatenations in one regular expression, since it is an associative operation. The same is true of union:

$L \cdot (L \cdot L) = (L \cdot L) \cdot L$
$L + (L + L) = (L + L) + L$

A word of explanation on nested Kleene *'s is in order. When a * operation occurs within another * operation, the two are independent. That is, in generating a sample string, each * generates 0 or more occurrences independently. For example, the regular expression (0*1)* could generate the string 0001101. The outer * repeats three times; the first time the inner * repeats three times, the second time the inner * repeats zero times, and the third time the inner * repeats once.

Sample Problem 2.0.2

For each of the following regular expressions, list six strings which are in its language.

1. *(a(b+c)*)*d*
2. *(a+b)*(c+d)*
3. *(a*b*)**

Solution:

1. *d ad abd acd aad abbcbd*
2. *c d ac abd babc bad*
3. *ϵ a b ab ba aa*
   *Note that (a*b*)* = (a+b)**

Sample Problem 2.0.3

Give a regular expression for each of the languages described in Sample Problem 2.0.2.

Solution:

1. `1*01*(01*01*)*`
2. `(0+1)*111(0+1)*`
3. `1*01*01*01*`
4. `(00+11)*(01+10)(1(0(11)*0)*1+0(1(00)*1)*0)*1(0(11)*0)* + (00+11)*0`

An algorithm for converting a finite state machine to an equivalent regular expression is beyond the scope of this text, but may be found in Hopcroft & Ullman [1979].

2.0.4 Exercises

1. Suppose L1 represents the set of all strings from the alphabet 0,1 which contain an even number of ones (even parity). Which of the following strings belong to L1?

   (a) 0101
   (b) 110211
   (c) 000
   (d) 010011
   (e) ϵ

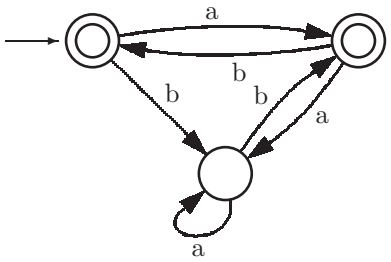
2. Suppose L2 represents the set of all strings from the alphabet a,b,c which contain an equal number of a's, b's, and c's. Which of the following strings belong to L2?

   (a) bca
   (b) accbab
   (c) ϵ
   (d) aaa
   (e) aabbcc

3. Which of the following are examples of languages?

   (a) L1 from Problem 1 above.
   (b) L2 from Problem 2 above.
   (c) Java
   (d) The set of all programming languages
   (e) Swahili

4. Which of the following strings are in the language specified by this finite state machine?



   (a) abab
   (b) bbb
   (c) aaab
   (d) aaa
   (e) ϵ

5. Show a finite state machine with input alphabet 0,1 which accepts any string having an odd number of 1's and an odd number of 0's.

6. Describe, in your own words, the language specified by each of the following finite state machines with alphabet a,b.

(a)

	a	b
A	B	A
B	B	C
C	B	D
*D	B	A

(b)

	a	b
A	B	A
B	B	C
C	B	D
*D	D	D

(c)

	a	b
*A	A	B
*B	C	B
C	C	C

(d)

	a	b
A	B	A
B	A	B
*C	C	B

(e)

	a	b
A	B	B
*B	B	B

7. Which of the following strings belong to the language specified by this regular expression: (a+bb)*a

   (a) ϵ
   (b) aaa
   (c) ba
   (d) bba
   (e) abba

8. Write regular expressions to specify each of the languages specified by the finite state machines given in Problem 6.

9. Construct finite state machines which specify the same language as each of the following regular expressions. The input alphabet in parts a,b,d is {a,b,c}. The input alphabet in part c is {a,b}. The input alphabet in part e is {a,b,c,d}.

   (a) (a+b)*c

   (b) (aa)*(bb)*c

   (c) (a*b*)*

   (d) (a+bb+c)a*

   (e) ((a+b)(c+d))*

10. Show a string of zeros and ones which is not in the language of the regular expression (0*1)*.

11. Show a finite state machine which accepts multiples of 3, expressed in binary (ϵ is excluded from this language).

2.1 Lexical Tokens

The first phase of a compiler is called lexical analysis. Because this phase scans the input string without backtracking (i.e. by reading each symbol once, and processing it correctly), it is often called a lexical scanner. As implied by its name, lexical analysis attempts to isolate the words in an input string. We use the word *word* in a technical sense. A word, also known as a lexeme, a lexical item, or a lexical token, is a string of input characters which is taken as a unit and passed on to the next phase of compilation. Examples of words are:

1. `while, if, else, for, ...` These are words which may have a particular predefined meaning to the compiler, as opposed to identifiers which have no particular meaning. Reserved words are keywords which are not available to the programmer for use as identifiers. In most programming languages, such as Java and C, all keywords are reserved. PL/1 is an example of a language which has key words but no reserved words.

2. identifiers - words that the programmer constructs to attach a name to a construct, usually having some indication as to the purpose or intent of the construct. Identifiers may be used to identify variables, classes, constants, functions, etc.

3. operators - symbols used for arithmetic, character, or logical operations, such as +,-,=,!=, etc. Notice that operators may consist of more than one character.

4. numeric constants - numbers such as 124, 12.35, 0.09E-23, etc. These must be converted to a numeric format so that they can be used in arithmetic operations, because the compiler initially sees all input as a string of characters. Numeric constants may be stored in a table.

5. character constants - single characters or strings of characters enclosed in quotes.

6. special characters - characters used as delimiters such as .,(,),,,;. These are generally single-character words.

7. comments - Though comments must be detected in the lexical analysis phase, they are not put out as tokens to the next phase of compilation.

8. white space - Spaces and tabs are generally ignored by the compiler, except to serve as delimiters in most languages, and are not put out as tokens.

9. newline - In languages with free format, newline characters should also be ignored, otherwise a newline token should be put out by the lexical scanner.

An example of Java source input, showing the word boundaries and types is given below:

```
while ( x33 <= 2.5e+33 - total ) calc ( x33 )  ;  //!

 1    6  2  3    4     3   2   6  2   6  2  6  6
```

During lexical analysis, a symbol table is constructed as identifiers are encountered. This is a data structure which stores each identifier once, regardless of the number of times it occurs in the source program. It also stores information about the identifier, such as the kind of identifier and where associated run-time information (such as the value assigned to a variable) is stored. This

data structure is often organized as a binary search tree, or hash table, for efficiency in searching.

When compiling block structured languages such as Java, C, or Algol, the symbol table processing is more involved. Since the same identifier can have different declarations in different blocks or procedures, both instances of the identifier must be recorded. This can be done by setting up a separate symbol table for each block, or by specifying block scopes in a single symbol table. This would be done during the parse or syntax analysis phase of the compiler; the scanner could simply store the identifier in a string space array and return a pointer to its first character.

Numeric constants must be converted to an appropriate internal form. For example, the constant *3.4e+6* should be thought of as a string of six characters which needs to be translated to floating point (or fixed point integer) format so that the computer can perform appropriate arithmetic operations with it. As we will see, this is not a trivial problem, and most compiler writers make use of library routines to handle this.

The output of this phase is a stream of tokens, one token for each word encountered in the input program. Each token consists of two parts: (1) a class indicating which kind of token and (2) a value indicating which member of the class. The above example might produce the following stream of tokens:

```
Token   Token
Class   Value

1       [code for while]
6       [code for (]
2       [ptr to symbol table entry for x33]
3       [code for <=]
4       [ptr to constant table entry for 2.5e+33]
3       [code for -]
2       [ptr to symbol table entry for total]
6       [code for )]
2       [ptr to symbol table entry for calc]
6       [code for (]
2       [ptr to symbol table entry for x33]
6       [code for )]
6       [code for ;]
```

Note that the comment is not put out. Also, some token classes might not have a value part. For example, a left parenthesis might be a token class, with no need to specify a value.

Some variations on this scheme are certainly possible, allowing greater efficiency. For example, when an identifier is followed by an assignment operator, a single assignment token could be put out. The value part of the token would

be a symbol table pointer for the identifier. Thus the input string $x =$, would be put out as a single token, rather than two tokens. Also, each keyword could be a distinct token class, which would increase the number of classes significantly, but might simplify the syntax analysis phase.

Note that the lexical analysis phase does not check for proper syntax. The input could be
} `while if ( {` and the lexical phase would put out five tokens corresponding to the five words in the input. (Presumably the errors will be detected in the syntax analysis phase.)

If the source language is not case sensitive, the scanner must accommodate this feature. For example, the following would all represent the same keyword: then, tHeN, Then, THEN. A preprocessor could be used to translate all alphabetic characters to upper (or lower) case. Java is case sensitive.

2.1.1 Exercises

1. For each of the following Java input strings show the word boundaries and token classes (for those tokens which are not ignored) selected from the list in Section 2.1.

```
(a)         for (i=start; i<=fin+3.5e6; i=i*3)
                      ac=ac+/*incr*/1;
(b)         { ax= 33 bx=/*if*/31.4 } // ax + 3;
(c)         if/*if*/a)}+whiles
```

2. Since Java is free format, newline characters are ignored during lexical analysis (except to serve as white space delimiters and to count lines for diagnostic purposes). Name at least two high-level programming languages for which newline characters would not be ignored for syntax analysis.

3. Which of the following will cause an error message from your Java compiler?

```
(a) A comment inside a quoted string:
"this is /*not*/ a comment"

(b) A quoted string inside a comment
/*this is "not" a string*/

(c) A comment inside a comment
/*this is /*not*/ a comment*/

(d) A quoted string inside a quoted string
"this is "not" a string"
```

4. Write a Java method to sum the codes of the characters in a given String:

```
public int sum (String s)
{ ... }
```

2.2 Implementation with Finite State Machines

Finite state machines can be used to simplify lexical analysis. We will begin by looking at some examples of problems which can be solved easily with finite state machines. Then we will show how actions can be included to process the input, build a symbol table, and provide output.

A finite state machine can be implemented very simply by an array in which there is a row for each state of the machine and a column for each possible input symbol. This array will look very much like the table form of the finite state machine shown in Figure 2.3. It may be necessary or desirable to code the states and/or input symbols as integers, depending on the implementation programming language. Once the array has been initialized, the operation of the machine can be easily simulated, as shown below:

```
boolean [] accept = new boolean [STATES];
int [][] fsm = new int[STATES][INPUTS];         // state table
// initialize table here...
int inp = 0;                                    // input symbol (0..INPUTS)
int state = 0;                                  // starting state;
try
{  inp = System.in.read() - '0'; // character input,
   // convert to int.
   while (inp>=0 && inp<INPUTS)
   {  state = fsm[state][inp];                  // next state
      inp = System.in.read() - '0';             // get next input
   }
} catch (IOException ioe)
  {  System.out.println ("IO error " + ioe); }

if (accept[state])
   System.out.println ("Accepted"); System.out.println ("Rejected");
```

When the loop terminates, the program would simply check to see whether the state is one of the accepting states to determine whether the input is accepted. This implementation assumes that all input characters are represented by small integers, to be used as subscripts of the array of states.

2.2.1 Examples of Finite State Machines for Lexical Analysis

An example of a finite state machine which accepts any identifier beginning with a letter and followed by any number of letters and digits is shown in

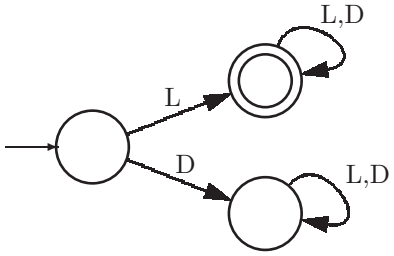


Figure 2.4: Finite state machine to accept identifiers

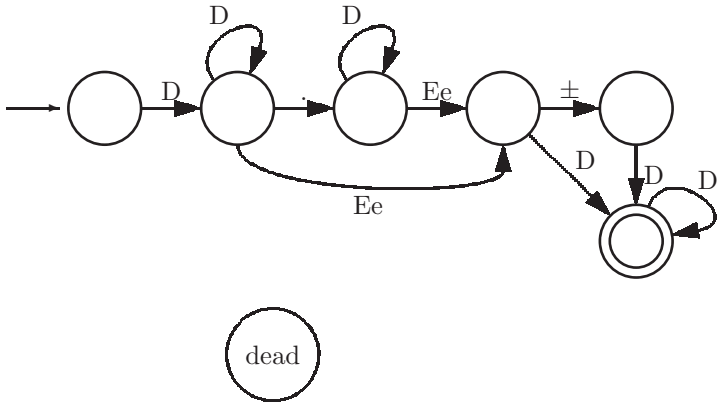


Figure 2.5: Finite state machine to accept numeric constants (all unspecified transitions are to the dead state)

Figure 2.4. The letter 'L' represents any letter (a-z), and the letter 'D' represents any numeric digit (0-9). This implies that a preprocessor would be needed to convert input characters to tokens suitable for input to the finite state machine.

A finite state machine which accepts numeric constants is shown in Figure 2.5.

Note that these constants must begin with a digit, and numbers such as .099 are not acceptable. This is the case in some languages, such as Pascal, whereas Java does permit constants which do not begin with a digit. We could have included constants which begin with a decimal point, but this would have required additional states.

A third example of the use of state machines in lexical analysis is shown in Figure 2.6. This machine accepts the keywords *if, int, import, for, float.* This machine is not completely specified, because in order for it to be used in a compiler it would have to accommodate identifiers as well as keywords. In particular, identifiers such as *i, wh, fo*, which are prefixes of keywords, and identifiers such as *fork*, which contain keywords as prefixes, would have to be handled. This problem will be discussed below when we include actions in the finite state machine.

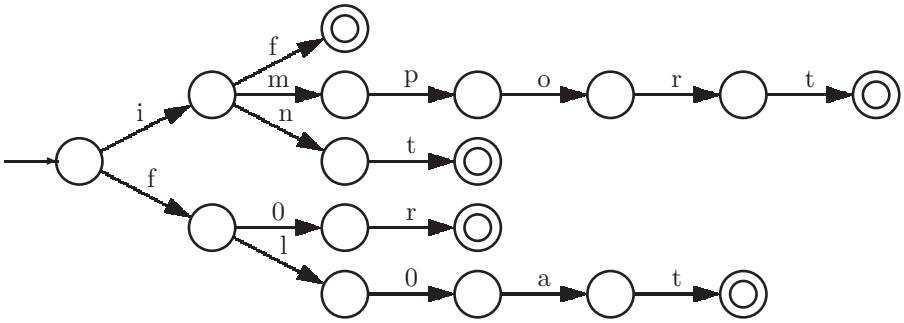


Figure 2.6: Finite state machine to accept key words

```
void p()
{  if (parity==0)  parity = 1;
   else  parity = 0;
}
```

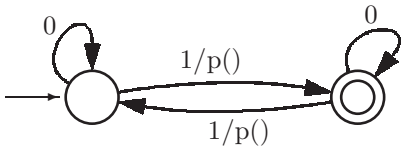


Figure 2.7: Parity Bit Generator

2.2.2 Actions for Finite State Machines

At this point, we have seen how finite state machines are capable of specifying a language and how they can be used in lexical analysis. But lexical analysis involves more than simply recognizing words. It may involve building a symbol table, converting numeric constants to the appropriate data type, and putting out tokens. For this reason, we wish to associate an action, or function to be invoked, with each state transition in the finite state machine.

This can be implemented with another array of the same dimension as the state transition array, which would be an arrray of functions to be called as each state transition is made. For example, suppose we wish to put out keyword tokens corresponding to each of the keywords recognized by the machine of Figure 2.6. We could associate an action with each state transition in the finite state machine. Moreover, we could recognize identifiers and call a function to store them in a symbol table.

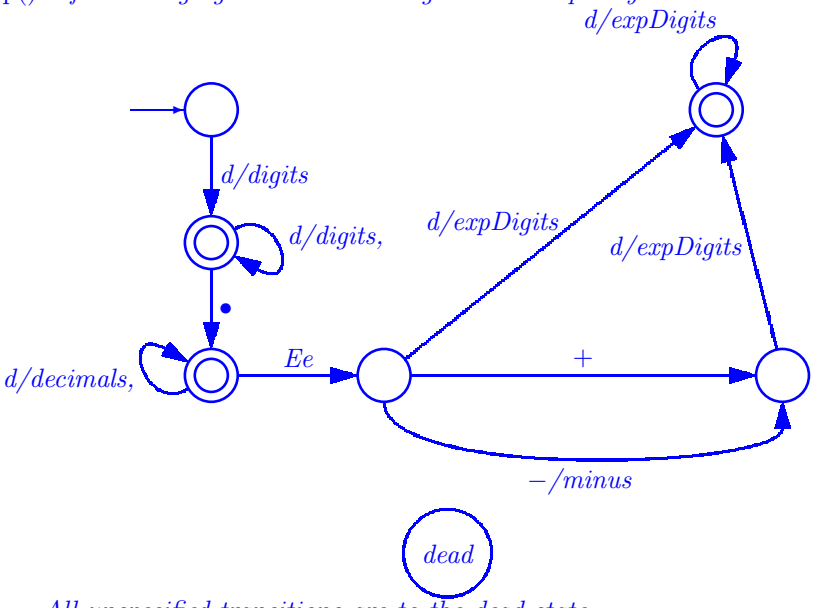
In Figure 2.7 we show an example of a finite state machine with actions. The purpose of the machine is to generate a parity bit so that the input string and parity bit will always have an even number of ones. The parity bit, *parity*, is initialized to 0 and is complemented by the function *p()*.

Sample Problem 2.2.1

Design a finite state machine, with actions, to read numeric strings and convert them to an appropriate internal format, such as floating point.

Solution:

In the state diagram shown below we have included method calls designated digits(), decimals(), minus(), and expDigits() which are to be invoked as the corresponding transition occurs. In other words, a transition marked i/p *means that if the input is* i*, invoke method* p() *before changing state and reading the next input symbol.*



All unspecified transitions are to the dead state.

The methods referred to in the state diagram are shown below:

```
// instance variables
int d;                  // A single digit, 0..9
int places=0;           // places after the decimal point
int mantissa=0;         // all the digits in the number
int exp=0;              // exponent value
int signExp=+1;         // sign of the exponent
```

```
// process digits before the decimal point
void digits()
{  mantissa = mantissa * 10 + d;   }

// process digits after the decimal point
void decimals()
{  digits();
   places++;             // count places after the decimal point
}

// Change the sign of the exponent
void minus()
{  signExp = -1;  }

// process digits after the E
void expDigits()
{  exp = exp*10 + d;  }
```

The value of the numeric constant may then be computed as follows:

```
double result = mantissa * Math.pow (10, signExp * exp - places);
```

where $Math.pow(x, y) = x^y$

2.2.3 Exercises

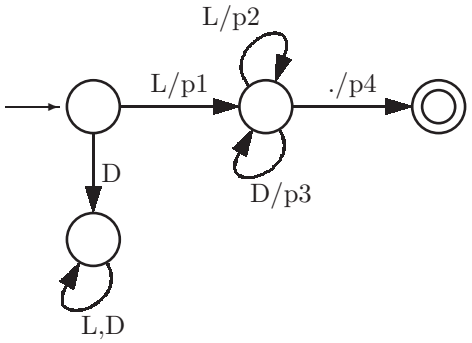
1. Show a finite state machine which will recognize the words RENT, RENEW, RED, RAID, RAG, and SENT. Use a different accepting state for each of these words.

2. Modify the finite state machine of Figure 2.5 to include numeric constants which begin with a decimal point and have digits after the decimal point, such as .25, without excluding any constants accepted by that machine.

3. Add actions to your solution to the previous problem so that numeric constants will be computed as in Sample Problem **??**.

4. Show a finite state machine that will accept C-style comments /* as shown here */. Use the symbol A to represent any character other than * or /; thus the input alphabet will be {/,*,A}.

5. What is the output of the finite state machine, below, for each of the following inputs (L represents any letter, and D represents any numeric digit; also, assume that each input is terminated with a period):




```
int sum;
void p1()                                   void p2()
{  sum = L;  }                               {  sum += L;  }

void p3()                                   int hash (int n)
{  sum += D;  }                             {  return n % 10;  }

void p4()
{ System.out.println(hash(sum));  }

(a) ab3.
(b) xyz.
(c) a49.
```

6. Show the values that will be asigned to the variable *mantissa* in Sample Problem **??** as the input string 46.73e-21 is read.

2.3 Lexical Tables

One of the most important functions of the lexical analysis phase is the creation of tables which are used later in the compiler. Such tables could include a symbol table for identifiers, a table of numeric constants, string constants, statement labels, and line numbers for languages such as Basic. The implementation techniques discussed below could apply to any of these tables.

2.3.1 Sequential Search

The table could be organized as an array or linked list. Each time a word is encountered, the list is scanned and if the word is not already in the list, it is

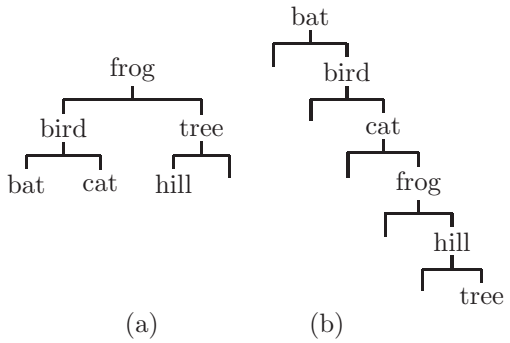


Figure 2.8: (a) A balanced binary search tree and (b) a binary search tree which is not balanced

added at the end. As we learned in our data structures course, the time required to build a table of n words is $O(n^2)$. This sequential search technique is easy to implement but not very efficient, particularly as the number of words becomes large. This method is generally not used for symbol tables, or tables of line numbers, but could be used for tables of statement labels, or constants.

2.3.2 Binary Search Tree

The table could be organized as a binary tree having the property that all of the words in the left subtree of any word precede that word (according to a sort sequence), and all of the words in the right subtree follow that word. Such a tree is called a *binary search tree*. Since the tree is initially empty, the first word encountered is placed at the root. Each time a word, w, is encountered the search begins at the root; w is compared with the word at the root. If w is smaller, it must be in the left subtree; if it is greater, it must be in the right subtree; and if it is equal, it is already in the tree. This is repeated until w has been found in the tree, or we arrive at a leaf node not equal to w, in which case w must be inserted at that point. Note that the structure of the tree depends on the sequence in which the words were encountered as depicted in Figure 2.8, which shows binary search trees for (a) frog, tree, hill, bird, bat, cat and for (b) bat, bird, cat, frog, hill, tree. As you can see, it is possible for the tree to take the form of a linked list (in which case the tree is said not to be balanced). The time required to build such a table of n words is $O(n \log n)$ in the best case (the tree is balanced), but could be $O(n^2)$ in the worst case (the tree is not balanced).

The student should bear in mind that each word should appear in the table only once, regardless how many times it may appear in the source program. In a later chapter we will see how the symbol table is used and what additional information is stored in it.

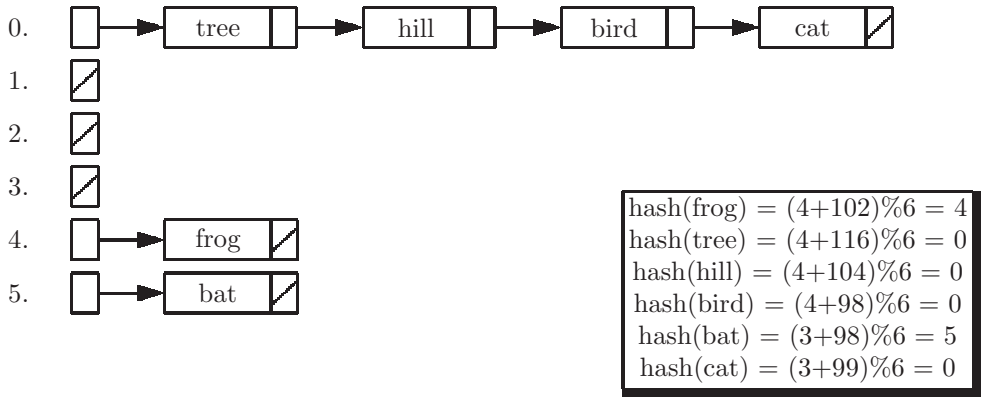


Figure 2.9: Hash table corresponding to the words entered for Figure 2.8 (a)

2.3.3 Hash Table

A hash table can also be used to implement a symbol table, a table of constants, line numbers, etc. It can be organized as an array, or as an array of linked lists, which is the method used here. We start with an array of null pointers, each of which is to become the head of a linked list. A word to be stored in the table is added to one of the lists. A hash function is used to determine which list the word is to be stored in. This is a function which takes as argument the word itself and returns an integer value which is a valid subscript to the array of pointers. The corresponding list is then searched sequentially, until the word is found already in the table, or the end of the list is encountered, in which case the word is appended to that list. The selection of a good hash function is critical to the efficiency of this method. Generally, we will use some arithmetic combination of the letters of the word, followed by dividing by the size of the hash table and taking the remainder. An example of a hash function would be to add the length of the word to the ascii code of the first letter and take the remainder on division by the array size, so that hash(bird) = (4+98) size of the array of pointers. The resulting value will always be in the range 0..HASHMAX-1 and can be used as a subscript to the array. Figure 2.9 depicts the hash table corresponding to the words entered for Figure 2.8 (a), where the value of HASHMAX is 6. Note that the structure of the table does not depend on the sequence in which the words are encountered (though the sequence of words in a particular list could vary).

2.3.4 Exercises

1. Show the binary search tree which would be constructed to store each of the following lists of identifiers:

(a) minsky, babbage, turing, ada, boole, pascal, vonneuman

(b) ada, babbage, boole, minsky, pascal, turing, vonneuman

(c) sum, x3, count, x210, x, x33

2. Show how many string comparisons would be needed to store a new identifier in a symbol table organized as a binary search tree containing:

   (a) 2047 identifiers, and perfectly balanced

   (b) 2047 identifiers which had been entered inalphabetic order (worst case)

   (c) $2^n - 1$ identifiers, perfectly balanced

   (d) n identifers, and perfectly balanced

3. Write a program in Java which will read a list of words from the keyboard, one word per line. If the word has been entered previously, the output should be OLD WORD. Otherwise the output should be NEW WORD. Use the following declaration to implement a binary search tree to store the words.

```
class Node
{  Node left;
   String data;
   Node right;

   public Node (String s)
   {  left = right = null;
      data = s
   }
}
Node bst;
```

4. Many textbooks on data structures implement a hash table as an array of words to be stored, whereas we suggest implementing with an array of linked lists. What is the main advantage of our method? What is the main disadvantage of our method?

5. Show the hash table which would result for the following identifiers using the example hash function of Section 2.3.3: bog, cab, bc, cb, h33, h22, cater.

6. Show a single hash function for a hash table consisting of ten linked lists such that none of the word sequences shown below causes a single collision.

   (a) ab, ac, ad, ae

   (b) ae, bd, cc, db

   (c) aa, ba, ca, da

7. Show a sequence of four identifiers which would cause your hash function defined in the previous problem to generate a collision for each identifier after the first.

2.4 Lexical Analysis with SableCC

The Unix programming environment includes several utility programs which are intended to improve the programmer's productivity. Included among these utilities are lex, for lexical analysis, and yacc (yet another compiler-compiler), for syntax analysis. These utilities generate compilers from a set of specifications and are designed to be used with the C programming language. When Java became popular, several replacements for lex and yacc became freely available on the internet: JLex, for lexical analysis; CUP (Constructor of Useful Parsers); ANTLR (Another Tool for Language Recognition); JavaCC, from Sun Microsystems; and SableCC, from McGill University. Of these, JavaCC is probably the most widely used. However, SableCC has several advantages over JavaCC:

- SableCC is designed to make good use of the advantages of Java; it is object-oriented and makes extensive use of class inheritance.
- With SableCC compilation errors are easier to fix.
- SableCC generates modular software, with each class in a separate file.
- SableCC generates syntax trees, from which atoms or code can be generated.
- SableCC can accommodate a wider class of languages than JavaCC (which permits only LL(1) grammars).

For these reasons we will be using SableCC, though all of our examples can also be done using JavaCC, JLex, or ANTLR.

Unlike the lex/yacc utilities which could be used separately or together for lexical and syntax analysis, respectively, SableCC combines lexical and syntax analysis into a single program. Since we have not yet discussed syntax analysis, and we wish to run SableCC for lexical analysis, we provide a SableCC template for the student to use. For more documentation on SableCC visit `http://www.sablecc.org`.

2.4.1 SableCC Input File

The input to SableCC consists of a text file, named with a .grammar suffix, with six sections; we will use only the first four of these sections in this chapter:

1. Package declaration
2. Helper declarations
3. States declarations

4. Token declarations
5. Ignored tokens
6. Productions

At the present time the student may ignore the Ignored tokens and Productions sections, whereas the sections on Helper declarations, States declarations, and Token declarations are relevant to lexical analysis. The required Java classes will be provided to the student as a standard template. Consequently, the input file, named language.grammar will be arranged as shown below:

```
Package package-name ;

Helpers
[ Helper declarations, if any, go here ]

States
[ State declarations, if any, go here ]

Tokens
[ Token declarations go here ]
```

Helpers, States, and Tokens will be described in the following sub-sections, although not in that sequence. All names, whether they be Helpers, States, or Tokens should be written using lower case letters and underscore characters. In any of these sections, single-line comments, beginning with //, or multi-line comments, enclosed in /* .. */ may be used.

2.4.1.1 Token Declarations

All lexical tokens in SableCC must be declared (given a name) and defined using the operations described here. These tokens are typically the "words" which are to be recognized in the input language, such as numbers, identifiers, operators, keywords, ... A Token declaration takes the form:

Token-name = Token-definition ;

For example: `left_paren = '(' ;`

A Token definition may be any of the following:

- A character in single quotes, such as `'w', '9', or '$'`.
- A number, written in decimal or hexadecimal, matches the character with that ascii (actually unicode) code. Thus, the number 13 mathches a newline character (the character `'\n'` works as well).
- A set of characters, specified in one of the following ways:
  - A single quoted character qualifies as a set consisting of one character.

  - A range of characters, with the first and last placed in brackets:
  - `['a'..'z']  // all lower case letters`
  - `['0'..'9']  // all numeric characters`
  - `[9..99] // all characters whose codes are in the range 9 through 99,inclusive`
  - A union of two sets, specified in brackets with a plus as in `[set1 + set2]`. Example:

    ```
    [['a'..'z'] + ['A'..'Z']]  // matches any letter
    ```

  - A difference of two sets, specified in brackets with a minus as in `[set1 - set2]` This matches any character in set1 which is not also in set2. Example:

    ```
    [[0..127] - ['\t' + '\n']]
    // matches any ascii character
    // except tab and newline.
    ```

- A string of characters in single quotes, such as 'while'.

Regular expressions, with some extensions to the operators described in section 2.0.3, may also be used in token definitions. If p and q are token definitions, then so are:

- `(p)` parentheses may be used to determine the order of operations (precedence is as defined in section 2.0.3.
- `pq` the concatenation of two token definitions is a valid token definition.
- `p|q` the union of two token definitions (note the plus symbol has a different meaning).
- `p*` the closure (kleene *) is a valid token definition, matching 0 or more repetitions of p.
- `p+` similar to closure, matches 1 or more repetitions of the definition p.
- `p?` matches an optional p, i.e. 0 or 1 repetitions of the definition p.

Note the two distinct uses of the '+' symbol: If s1 and s2 are sets, [s1+s2] is their union. If p is a regular expression, p+ is also a regular expression. To specify union of regular expressions, use $'|'$. Some examples of token definitions are shown below:

```
number = ['0'..'9']+ ;              // A number is 1 or more
                                    // decimal digits.

// An identifier must begin with an alphabetic character:
identifier = [['a'..'z']+[['A'..'Z']]
                  (['a'..'z'] | ['A..'Z'] | ['0'..'9'] | '_')* ;
```

```
// Six relational operators:
rel_op = ['<' + '>'] '='? | '==' | '!=' ;
```

There are two important rules to remember when more than one token definition matches input characters at a given point in the input file:

- When two token definitions match the input, the one matching the longer input string is selected.
- When two token definitions match input strings of the same length, the token definition listed first is selected. For example, the following would not work as desired:

  ```
  Tokens
    identifier = ['a'..'z']+ ;
    keyword = 'while' | 'for' | 'class' ;
  ```

  An input of 'while' would be returned as an identifier, as would an input of 'whilex'.

  Instead the tokens should be defined as:

  ```
  Tokens
    keyword = 'while' | 'for' | 'class' ;
    identifier = ['a'..'z']+ ;
  ```

  With this definition, the input 'whilex' would be returned as an identifier, because the keyword definition matches 5 characters, 'while', and the identifier definition matches 6 characters, 'whilex'; the longer match is selected. The input 'while' would be a keyword; since it is matched by two definitions, SableCC selects the first one, keyword.

2.4.1.2 Helper Declarations

The definition of identifier, above, could have been simplified with a macro capability. Helpers are permitted for this purpose. Any helper which is defined in the Helpers section may be used as part of a token defnition in the Tokens section. For example, we define three helpers below to facilitate the definitions of number, identifier, and space:

```
Helpers
  digit = ['0'..'9'] ;
  letter = [['a'..'z'] + ['A'..'Z']] ;
  sign = '+' | '-' ;
  newline = 10 | 13 ;                // ascii codes
```

```
  tab = 9 ;                          // ascii code for tab

Tokens
  number = sign? digit+ ;          // A number is an optional
                                  // sign, followed by 1 or more digits.

// An identifier is a letter followed by 0 or more letters,
// digits, or underscores:
  identifier = letter (letter | digit | '_')* ;
  space = ' ' | newline | tab ;
```

Students who may be familiar with macros in the unix utility lex will see an important distinction here. Whereas in lex, macros are implemented as textual substitutions, in SableCC helpers are implemented as semantic substitutions. For example, the definition of number above, using lex would be obtained by substituting directly the definition of sign into the definition of number:

```
number =  sign? digit+
       =  '+' | '-'? ['0'..'9']+
       =  '+' | ('-'? ['0'..'9']+)
```

This says that a number is either a plus or an optional minus followed by one or more digits, which is not what the user intended. We have seen many students trip on this stumbling block when using lex, which has finally been eliminated by the developers of SableCC.

Sample Problem 2.4.1

Show the sequence of tokens which would be recognized by the preceding definitions of number, identifier, *and* space *for the following input (also show the text which corresponds to each token):*

```
334 abc abc334
```

Solution:

number	*334*
space	
identifier	*abc*
space	
identifier	*abc334*

2.4.1.3 State Declarations, Left Context, and Right Context

For purposes of lexical analysis, it is often helpful to be able to place the lexical scanner in one or more different states as it reads the input (it is, after all, a finite state machine). For example, the input 'sum + 345' would normally be returned as three tokens: an identifier, an arithmetic operator, and a number. Suppose, however, that this input were inside a comment or a string:

```
// this is a comment sum + 345
```

In this case the entire comment should be ignored. In other words, we wish the scanner to go into a different state, or mode of operation, when it sees the two consecutive slashes. It should remain in this state until it encounters the end of the line, at which point it would return to the default state. Some other uses of states would be to indicate that the scanner is processing the characters in a string; the input character is at the beginning of a line; or some other left context, such as a '$' when processing a currency value. To use states, simply identify the names of the states as a list of names separated by commas in the States section:

```
States
  statename1, statename2, statename3,... ;
```

The first state listed is the start state; the scanner will start out in this state. In the Tokens section, any definition may be preceded by a list of state names and optional state transitions in curly braces. The definition will be applied only if the scanner is in the specified state:

```
{statename} token = def ;     // apply this definition only if the scanner is
                                // in state statename (and remain in that
                                // state)
```

How is the scanner placed into a particular state? This is done with the transition operator, $->$. A transition operator may follow any state name inside the braces:

```
{statename->newstate} token = def;
                      // apply this definition only if the scanner is in statename,
                      // and change the state to newstate.
```

A definition may be associated with more than one state:

```
{state1->state2, state3->state4, state5} token = def;
                // apply this definition only if the scanner is in state1
                // (change to state2), or if the scanner is in state3
                // (change to state4), or if the scanner is in state5
                // (remain in state5).
```

Definitions which are not associated with any states may be applied regardless of the state of the scanner:

```
token = def;    // apply this definition regardless of the current state of the
                // scanner.
```

The following example is taken from the SableCC web site. Its purpose is to make the scanner toggle back and forth between two states depending on whether it is at the beginning of a line in the input. The state bol represents beginning of line, and inline means that it is not at the beginning of a line. The end-of-line character may be just '\n', or 13, but on some systems it could be 10 (linefeed), or 10 followed by 13. For portability, this scanner should work on any of these systems.

```
States
  bol, inline;          // Declare the state names. bol is
                        // the start state.
Tokens
  {bol->inline, inline} char = [[0..0xffff] - [10 + 13]];
                    // Scanning a non-newline char.  Apply
                    // this in either state, New state is inline.
  {bol, inline->bol} eol = 10 | 13 | 10 13;
                    // Scanning a newline char.  Apply this in
                    // either state.  New state is bol.
```

In general, states can be used whenever there is a need to accommodate a left context for a particular token definition.

Sample Problem 2.4.2

Show the token and state definitions needed to process a text file containing numbers, currency values, and spaces. Currency values begin with a dollar sich, such as '$3045' and '$9'. Assume all numbers and currency values are whole numbers. Your definitions should be able to distinguish between currency values (money) and ordinary numbers (number). You may also use helpers.

Solution:

```
Helpers
  num = ['0'..'9']+ ;          // 1 or more digits

States
  def, currency;              // def is start state

Tokens
  space = (' ' | 10 | 13 | '\t') ;
  {def -> currency} dollar = '$' ;    // change to currency state
  {currency -> def} money = num;      // change to def state
  {def} number = num;                 // remain in def state
```

It is also possible to specify a right context for tokens. This is done with a forward slash ('/'). To recognize a particular token only when it is followed by a certain pattern, include that pattern after the slash. The token, not including the right context (i.e. the pattern), will be matched only if the right context is present. For example, if you are scanning a document in which all currency amounts are followed by DB or CR, you could match any of these with:

```
currency = number / space* 'DB' | number / space * 'CR' ;
```

In the text:

```
Your bill is 14.50 CR, and you are 12 days late.
```

SableCC would find a currency token as '14.50' (it excludes the ' CR' which is the right context). The '12' would not be returned as a currency token because the right context is not present.

2.4.1.4 Ignored Tokens

The Ignored Tokens section of the SableCC grammar file is optional. It provides the capability of declaring tokens that are ignored (not put out by the lexer). Typically things like comments and white space will be ignored. The declaration takes the form of a list of the ignored tokens, separated by commas, and ending with a semicolon, as shown in the following example:

```
Ignored Tokens
  space, comment ;
```

2.4.1.5 An Example of a SableCC Input File

Here we provide a complete example of a SableCC input file (a "grammar") along properly. The student should make modifications to the source code given here with two Java classes which need to be defined in order for it to execute in order to test other solutions on the computer. The example will produce a scanner which will recognize numbers (ints), identifiers, arithmetic operators, relational operators, and parentheses in the input file. We call this example "lexing", because it demonstrates how to generate a lexical scanner; the source code is placed in a file called lexing.grammar (we will learn about grammars in Chapter 3).

```
Package lexing ;          // A Java package is produced for the
                          // generated scanner

Helpers
 num = ['0'..'9']+;       // A num is 1 or more decimal digits
 letter = ['a'..'z'] | ['A'..'Z']  ;
                         // A letter is a single upper or
                         // lowercase character.

Tokens
 number = num;           // A number token is a whole number
 ident = letter (letter | num)* ;
                         // An ident token is a letter followed by
                         // 0 or more letters and numbers.
 arith_op = [ ['+' + '-' ] + ['*' + '/' ] ] ;
                         // Arithmetic operators
 rel_op = ['<' + '>'] | '==' | '<=' | '>=' | '!=' ;
                         // Relational operators
 paren = ['(' + ')'];    // Parentheses
 blank = (' ' | '\t' | 10 | '\n')+ ;     // White space
 unknown = [0..0xffff] ;
                         // Any single character which is not part
                         // of one of the above tokens.
```

2.4.2 Running SableCC

Before running SableCC, a class containing a main method must be defined. A sample of this class is shown below, and is available at cs.rowan.edu/∼bergmann/books. This Lexing class is designed to be used with the grammar shown above in section 2.4.1. Each token name is prefixed by a 'T', so you should modify the token names to conform to your own needs. A special token, EOF, represents the end of the input file.

```
package lexing;
```

```
import lexing.lexer.*;
import lexing.node.*;
import java.io.*;        // Needed for pushbackreader and
             // inputstream
class Lexing
{
static Lexer lexer;
static Object token;
public static void main(String [] args)
{
  lexer = new Lexer
     (new PushbackReader
          (new InputStreamReader (System.in), 1024));
  token = null;
  try
    {
     while ( ! (token instanceof EOF))
       {     token = lexer.next();     // read next token
             if (token instanceof TNumber)
                     System.out.print ("Number:       ");
             else if (token instanceof TIdent)
                     System.out.print ("Identifier:  ");
             else if (token instanceof TArithOp)
                     System.out.print ("Arith Op:    ");
             else if (token instanceof TRelOp)
                    System.out.print ("Relational Op: ");
             else if (token instanceof TParen)
                     System.out.print ("Parentheses  ");
             else if (token instanceof TBlank)  ;
                 // Ignore white space
            else if (token instanceof TUnknown)
                     System.out.print ("Unknown        ");
      if (! (token instanceof TBlank))
       System.out.println (token);   // print token as a string
          }
       }
     catch (LexerException le)
        {  System.out.println ("Lexer Exception " + le); }
     catch (IOException ioe)
        {  System.out.println ("IO Exception " +ioe); }
}
}
```

There is now a two-step process to generate your scanner. The first step is to generate the Java class definitions by running SableCC. This will produce a sub-directory, with the same name as the language being compiled. All the

generated java code is placed in this sub-directory. Invoke SableCC as shown below:

```
sablecc languagename.grammar
```

(The exact form of this system command could be different depending on how SableCC has been installed on your computer) In our example it would be:

```
sablecc lexing.grammar
```

The second step required to generate the scanner is to compile these Java classes. First. copy the Lexing.java file from the web site to your lexing sub-directory, and make any necessary changes. Then compile the source files from the top directory:

```
javac languagename/*.java
```

In our case this would be:

```
javac lexing/*.java
```

We have now generated the scanner in lexing.Lexing.class. To execute the scanner:

```
java languagename.Classname
```

In our case this would be:

```
java lexing.Lexing
```

This will read from the standard input file (keyboard) and should display tokens as they are recognized. Use the end-of-file character to terminate the input (ctrl-d for unix, ctrl-z for Windows/DOS). A sample session is shown below:

```
java lexing.Lexing
sum = sum + salary ;

Identifier:     sum
Unknown         =
Identifier:     sum
Arith Op:       +
Identifier:     salary
Unknown         ;
```

2.4.3 Exercises

1. Modify the given SableCC lexing.grammar file and lexing/Lexing.java file to recognize the following 7 token classes.

```
(1)     Identifier (begins with letter, followed by letters, digits, _)
(2)     Numeric constant (float or int)
(3)     =    (assignment)
(4)     Comparison operator (==  <  >  <=  >=  !=)
(5)     Arithmetic operator ( +  -  *  / )
(6)     String constant "inside double-quote marks"
(7)     Keyword ( if  else  while  do  for  class )
        Comments        /* Using this method */
                        // or this method, but don't print a token
                        //     class.
```

2. Show the sequence of tokens recognized by the following definitions for each of the input files below:

```
Helpers
  char = ['a'..'z'] ['0'..'9']? ;
Tokens
  token1 = char char ;
  token2 = char 'x' ;
  token3 = char+ ;
  token4 = ['0'..'9']+ ;
  space = ' ' ;
```

Input files:

```
(a)     a1b2c3
(b)     abc3 a123
(c)     a4x ab r2d2
```

2.5 Case Study: Lexical Analysis for Decaf

In this section we present a description of the lexical analysis phase for the subset of Java we call Decaf. This represents the first phase in our case study: a complete Decaf compiler. The lexical analysis phase is implemented in the Helpers and Tokens sections of the SableCC source file, which is shown in its entirety in Appendix B.2 (refer to the file decaf.grammar).

The Decaf case study is implemented as a two-pass compiler. The syntax and lexical phases are implemented with SableCC. The result is a file of atoms, and a file of numeric constants. These two files form the input for the code generator, which produces machine code for a simulated machine, called mini.

In this section we describe the first two sections of the SableCC source file for Decaf, which are used for lexical analysis.

The Helpers section, shown below, defines a few macros which will be useful in the Tokens section. A letter is defined to be any single letter, upper or lower case. A digit is any single numeric digit. A digits is a string of one or more digits. An exp is used for the exponent part of a numeric constant, such as `1.34e12i`. A newline is an end-of-line character (for various systems). A non_star is any unicode character which is not an asterisk. A non_slash is any unicode character which is not a (forward) slash. A non_star_slash is any unicode character except for asterisk or slash. The helpers non_star and non_slash are used in the description of comments. The Helpers section, with an example for each Helper, is shown below:

```
Helpers                                              // Examples
  letter =    ['a'..'z'] | ['A'..'Z'] ;              //   w
  digit =     ['0'..'9'] ;                           //   3
  digits =    digit+ ;                               //  2040099
  exp  =      ['e' + 'E'] ['+' + '-']? digits;       //  E-34
  newline =   [10 + 13] ;                            //  '\n'
  non_star =  [[0..0xffff] - '*'] ;                  //   /
  non_slash = [[0..0xffff] - '/'];                   //  *
  non_star_slash = [[0..0xffff] - ['*' + '/']];      //  $
```

States can be used in the description of comments, but this can also be done without using states. Hence, we will not have a States section in our source file.

The Tokens section, shown below, defines all tokens that are used in the definition of Decaf. All tokens must be named and defined here. We begin with definitions of comments; note that in Decaf, as in Java, there are two kinds of comments: (1) single line comments, which begin with '//' and terminate at a newline, and (2) multi-line comments, which begin with '/*' and end with '*/'. These two kinds of comments are called comment1 and comment2, respectively. The definition of comment2, for multi-line comments, was designed using a finite state machine model as a guide (see exercise 4 in section 2.2). Comments are listed with white space as Ignored Tokens, i.e. the parser never even sees these tokens.

A space is any white space, including tab (9) and newline (10, 13) characters. Each keyword is defined as itself. The keyword *class* is an exception; for some reason SableCC will not permit the use of class as a name, so it is shortened to clas. A language which is not case-sensitive, such as BASIC or Pascal, would require a different strategy for keywords. The keyword while could be defined as

```
while =  ['w' + 'W'] ['h' + 'H'] ['i' + 'I'] ['l' + 'L'] ['e' + 'E'] ;
```

Alternatively, a preprocessor could convert all letters (not inside strings) to lower case.

A compare token is any of the six relational operators. The arithmetic operators, parentheses, braces, brackets, comma, and semicolon are all given names; this is tedious but unavoidable with SableCC. An identifier token is defined to be a letter followed by 0 or more letters, digits, and underscores. A number is a numeric constant which may have a decimal point and/or an exponent part. This is where we use the Helper exp, representing the exponent part of a number. Any character which has not been matched as part of the above tokens is called a misc token, and will most likely cause the parser to report a syntax error. The Tokens section is shown below:

```
Tokens
  comment1 = '//' [[0..0xffff]-newline]* newline ;
  comment2 = '/*' non_star* '*'
                (non_star_slash non_star* '*'+)* '/' ;

  space = ' ' | 9 | newline ;            // 9 = tab
  clas = 'class' ;                       // key words (reserved)
  public = 'public' ;
  static = 'static' ;
  void = 'void' ;
  main = 'main' ;
  string = 'String' ;
  int = 'int' ;
  float = 'float' ;
  for = 'for' ;
  while = 'while' ;
  if = 'if' ;
  else = 'else' ;
  assign = '=' ;
  compare = '==' | '<' | '>' | '<=' | '>=' | '!=' ;
  plus = '+' ;
  minus = '-' ;
  mult = '*' ;
  div = '/' ;
  l_par = '(' ;
  r_par = ')' ;
  l_brace = '{' ;
  r_brace = '}' ;
  l_bracket = '[' ;
  r_bracket = ']' ;
  comma = ',' ;
  semi = ';' ;
  identifier = letter (letter | digit | '_')* ;
  number  =  (digits '.'? digits? | '.'digits) exp? ;
  misc = [0..0xffff] ;
```

This completes the description of the lexical analysis of Decaf. The imple-

mentation makes use of the Java class Hashtable to implement a symbol table and a table of numeric constants. This will be discussed further in Chapter 5 when we define the Translation class to be used with SableCC.

2.5.1 Exercises

1. Extend the SableCC source file for Decaf, decaf.grammar, to accommodate string constants and character constants (these files can be found at http://cs.rowan.edu/∼bergmann/books). For purposes of this exercise, ignore the section on productions. A string is one or more characters inside double-quotes, and a character constant is one character inside single-quotes (do not worry about escape-chars, such as ‘
n’). Here are some examples, with a hint showing what your lexical scanner should find:

INPUT	HINT
"A long string"	One string token
" Another 'c' string"	One string token
"one" 'x' "three"	A string, a char, a string
" // string "	A string, no comment
// A "comment"	A comment, no string

2. Extend the SableCC source file decaf.grammar given at www.rowan.edu/ bergmann/books to permit a *switch* statement and a *do while* statement in Decaf:

   1. SwitchStmt → switch (Expr) { CaseList }
   2. CaseList → case NUM : StmtList
   3. CaseList → case default: StmtList
   4. CaseList → case NUM : StmtList CaseList
   5. Stmt → break ;
   6. DoStmt → do Stmt while (Expr)

3. Revise the token definition of the number token in decaf.grammar to exclude numeric constants which do not begin with a digit, such as .25 and .03e-4. Test your solution by running the software.

4. Rather than having a separate token class for each Decaf keyword, the scanner could have a single class for all keywords. Show the changes needed in the file decaf.grammar to do this.

2.6 Chapter Summary

This chapter on Lexical Analysis began with some introductory theory of formal languages and automata. A language, defined as a set of strings, is a vital

concept in the study of programming languages and compilers. An automaton is a theoretic machine, introduced in this chapter with finite state machines. It was shown how these theoretic machines can be used to specify programming language elements such as identifiers, constants, and keywords. We also introduced the concept of regular expressions, which can be used to specify the same language elements. Regular expressions are useful not only in lexical analysis, but also in utility programs and editors such as awk, ed, and grep, in which it is necessary to specify search patterns.

We then discussed the problem of lexical analysis in more detail, and showed how finite state machine theory can be used to implement a lexical scanner. The lexical scanner must determine the word boundaries in the input string. The scanner accepts as input the source program, which is seen as one long string of characters. Its output is a stream of tokens, where each token consists of a class and possibly a value. Each token represents a lexical entity, or word, such as an identifier, keyword, constant, operator, or special character.

A lexical scanner can be organized to write all the tokens to a file, at which point the syntax phase is invoked and reads from the beginning of the file. Alternatively, the scanner can be called as a subroutine to the syntax phase. Each time the syntax phase needs a token it calls the scanner, which reads just enough input characters to produce a single token to be returned to the syntax phase.

We also showed how a lexical scanner can create tables of information, such as a symbol table, to be used by subsequent phases of the compiler.

We introduced a compiler generator, SableCC, which includes a provision for generating a lexical scanner, using regular expressions to specify patterns to match lexical tokens in the source language. The SableCC source file consists of three sections relevant to lexical analysis: (1) Helpers (i.e. macros); (2) States; and (3) Tokens. We concluded the chapter with a look at a SableCC program which implements the lexical scanner for our case study: Decaf.

Chapter 3

Syntax Analysis

The second phase of a compiler is called syntax analysis. The input to this phase consists of a stream of tokens put out by the lexical analysis phase. They are then checked for proper syntax, i.e. the compiler checks to make sure the statements and expressions are correctly formed. Some examples of syntax errors in Java are:

```
x = (2+3) * 9); // mismatched parentheses

if x>y x = 2; // missing parentheses

while (x==3) do f1();  // invalid keyword do
```

When the compiler encounters such an error, it should put out an informative message for the user. At this point, it is not necessary for the compiler to generate an object program. A compiler is not expected to guess the intended purpose of a program with syntax errors. A good compiler, however, will continue scanning the input for additional syntax errors.

The output of the syntax analysis phase (if there are no syntax errors) could be a stream of atoms or syntax trees. An atom is a primitive operation which is found in most computer architectures, or which can be implemented using only a few machine language instructions. Each atom also includes operands, which are ultimately converted to memory addresses on the target machine. A syntax tree is a data structure in which the interior nodes represent operations, and the leaves represent operands, as discussed in Section 1.2.2. We will see that the parser can be used not only to check for proper syntax, but to produce output as well. This process is called syntax directed translation.

Just as we used formal methods to specify and construct the lexical scanner, we will do the same with syntax analysis. In this case however, the formal methods are far more sophisticated. Most of the early work in the theory of compiler design focused on syntax analysis. We will introduce the concept of a

formal grammar not only as a means of specifying the programming language, but also as a means of implementing the syntax analysis phase of the compiler.

3.0 Grammars, Languages, and Pushdown Machines

Before we discuss the syntax analysis phase of a compiler, there are some concepts of formal language theory which the student must understand. These concepts play a vital role in the design of the compiler. They are also important for the understanding of programming language design and programming in general.

3.0.1 Grammars

Recall our definition of language from Chapter 2 as a set of strings. We have already seen two ways of formally specifying a language: regular expressions and finite state machines. We will now define a third way of specifying languages, i.e. by using a grammar. A grammar is a list of rules which can be used to produce or generate all the strings of a language, and which does not generate any strings which are not in the language. More formally a grammar consists of:

1. A finite set of characters, called the input alphabet, the input symbols, or terminal symbols.

2. A finite set of symbols, distinct from the terminal symbols, called nonterminal symbols, exactly one of which is designated the starting nonterminal (if no nonterminal is explicitly designated as the starting nonterminal, it is assumed to be the nonterminal defined in the first rule).

3. A finite list of rewriting rules, also called productions, which define how strings in the language may be generated. Each of these rewriting rules is of the form $a \rightarrow b$, where a and b are arbitrary strings of terminals and nonterminals, and a is not null.

The grammar specifies a language in the following way: beginning with the starting nonterminal, any of the rewriting rules are applied repeatedly to produce a sentential form, which may contain a mix of terminals and nonterminals. If at any point, the sentential form contains no nonterminal symbols, then it is in the language of this grammar. If G is a grammar, then we designate the language specified by this grammar as L(G).

A derivation is a sequence of rewriting rules, applied to the starting nonterminal, ending with a string of terminals. A derivation thus serves to demonstrate that a particular string is a member of the language. Assuming that the starting nonterminal is S, we will write derivations in the following form:

$S \Rightarrow a \Rightarrow b \Rightarrow g \Rightarrow ... \Rightarrow x$

where a, b, g are strings of terminals and/or nonterminals, and x is a string of terminals.

In the following examples, we observe the convention that all lower case letters and numbers are terminal symbols, and all upper case letters (or words which begin with an upper case letter) are nonterminal symbols. The starting nonterminal is always S unless otherwise specified. Each of the grammars shown in this chapter will be numbered (G1, G2, G3, ...) for reference purposes. The first example is grammar G1, which consists of four rules, the terminal symbols {0,1}, and the starting nonterminal, S.

G1:

1. S $\rightarrow$ 0S0
2. S $\rightarrow$ 1S1
3. S $\rightarrow$ 0
4. S $\rightarrow$ 1

An example of a derivation using this grammar is:

$S \Rightarrow 0S0 \Rightarrow 00S00 \Rightarrow 001S100 \Rightarrow 0010100$

Thus, 0010100 is in L(G1), i.e. it is one of the strings in the language of grammar G1. The student should find other derivations using G1 and verify that G1 specifies the language of palindromes of odd length over the alphabet {0,1}. A palindrome is a string which reads the same from left to right as it does from right to left.

L(G1) = {0, 1, 000, 010, 101, 111, 00000, ... }

In our next example, the terminal symbols are {a,b} (ϵ represents the null string and is not a terminal symbol).

G2:

1. S $\rightarrow$ ASB
2. S $\rightarrow \epsilon$
3. A $\rightarrow$ a
4. B $\rightarrow$ b

$S \Rightarrow ASB \Rightarrow AASBB \Rightarrow AaSBB \Rightarrow AaBB \Rightarrow AaBb \Rightarrow Aabb \Rightarrow aabb$

Thus, aabb is in L(G2). G2 specifies the set of all strings of a's and b's which contain the same number of a's as b's and in which all the a's precede all the b's. Note that the null string is permitted in a rewriting rule.

$L(G2) = \{\epsilon, ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb, ...\} = \{a^n b^n\}$ such that $n \geq 0$

This language is the set of all strings of a's and b's which consist of zero or more a's followed by exactly the same number of b's.

Two grammars, g1 and g2, are said to be equivalent if L(g1) = L(g2); i.e., they specify the same language. In this example (grammar G2) there can be several different derivations for a particular string, i.e. the rewriting rules could have been applied in a different sequence to arrive at the same result.

Sample Problem 3.0.1

Show three different derivations using the grammar shown below:

1. $S \rightarrow a\ S\ A$
2. $S \rightarrow B\ A$
3. $A \rightarrow a\ b$
4. $B \rightarrow b\ A$

Solution:

$S \Rightarrow a\ S\ A \Rightarrow a\ B\ A\ A \Rightarrow a\ B\ a\ b\ A \Rightarrow a\ B\ a\ b\ a\ b \Rightarrow a\ b\ A\ a\ b\ a\ b \Rightarrow a\ b\ a\ b\ a\ b\ a\ b$

$S \Rightarrow a\ S\ A \Rightarrow a\ S\ a\ b \Rightarrow a\ B\ A\ a\ b \Rightarrow a\ b\ A\ A\ a\ b \Rightarrow a\ b\ a\ b\ A\ a\ b \Rightarrow a\ b\ a\ b\ a\ b\ a\ b$

$S \Rightarrow B\ A \Rightarrow b\ A\ A \Rightarrow b\ a\ b\ A \Rightarrow b\ a\ b\ a\ b$

Note that in the solution to this problem we have shown that it is possible to have more than one derivation for the same string: **ababab***.*

3.0.2 Classes of Grammars

In 1959 Noam Chomsky, a linguist, suggested a way of classifying grammars according to complexity [7]. The convention used below, and in the remaining chapters, is that the term *string* includes the null string and that, in referring to grammars, the following symbols will have particular meanings:

$A, B, C, ...$	A single nonterminal
$a, b, c, ...$	A single terminal
$..., X, Y, Z$	A single terminal or nonterminal
$..., x, y, z$	A string of terminals
α, β, γ	A string of terminals and nonterminals

Here is Chomsky's classification of grammars:

0. Unrestricted: An unrestricted grammar is one in which there are no restrictions on the rewriting rules. Each rule may consist of an arbitrary string of terminals and nonterminals on both sides of the arrow (though e is permitted on the right side of the arrow only). An example of an unrestricted rule would be:

$SaB \rightarrow cS$

1. Context-Sensitive: A context-sensitive grammar is one in which each rule must be of the form:

$\alpha A \gamma \rightarrow \alpha \beta \gamma$

where each of α, β, and γ is any string of terminals and nonterminals (including ϵ), and A represents a single nonterminal. In this type of grammar, it is the nonterminal on the left side of the rule (A) which is being rewritten, but only if it appears in a particular context, α on its left and γ on its right. An example of a context-sensitive rule is shown below:

$SaB \rightarrow caB$

which is another way of saying that an S may be rewritten as a c, but only if the S is followed by aB (i.e. when S appears in that context). In the above example, the left context is null.

2. Context-Free: A context-free grammar is one in which each rule must be of the form:

$A \rightarrow \alpha$

where A represents a single nonterminal and α is any string of terminals and nonterminals. Most programming languages are defined by grammars of this type; consequently, we will focus on context-free grammars. Note that both grammars G1 and G2, above, are context-free. An example of a context-free rule is shown below:

$A \rightarrow aABb$

3. Right Linear: A right linear grammar is one in which each rule is of the form:

$A \rightarrow aB$

or

$A \rightarrow a$

where A and B represent nonterminals, and a represents a terminal. Right linear grammars can be used to define lexical items such as identifiers, constants, and keywords.

Note that every context-sensitive grammar is also in the unrestricted class. Every context-free grammar is also in the context-sensitive and unrestricted classes. Every right linear grammar is also in the context-free, context-sensitive, and unrestricted classes. This is represented by the diagram of Figure 3.1, which depicts the classes of grammars as circles. All points in a circle belong to the class of that circle.

A context-sensitive language is one for which there exists a context-sensitive grammar. A context-free language is one for which there exists a context-free grammar. A right linear language is one for which there exists a right linear grammar. These classes of languages form the same hierarchy as the corresponding classes of grammars.

We conclude this section with an example of a context-sensitive grammar which is not context-free.

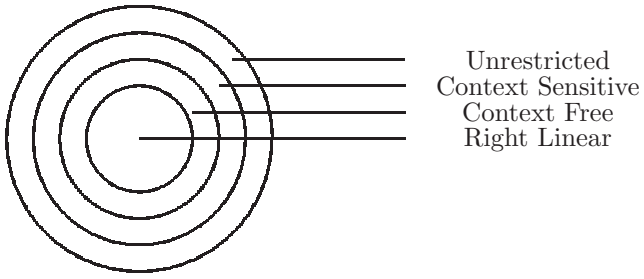


Figure 3.1: Classes of grammars

G3:

1. S $\rightarrow$ aSBC
2. S $\rightarrow \epsilon$
3. aB $\rightarrow$ ab
4. bB $\rightarrow$ bb
5. C $\rightarrow$ c
6. CB $\rightarrow$ CX
7. CX $\rightarrow$ BX
8. BX $\rightarrow$ BC

A derivation of the string *aabbcc* is shown below:

$S \Rightarrow aSBC \Rightarrow aaSBCBC \Rightarrow aaBCBC \Rightarrow aaBCXC \Rightarrow aaBBXC \Rightarrow aaBBCC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbCc \Rightarrow aabbcc$

The student should perform other derivations to understand that

$L(G3) = \{\epsilon, abc, aabbcc, aaabbbccc, ...\} = \{a^n b^n c^n\} where n \geq 0$

i.e., the language of grammar G3 is the set of all strings consisting of a's followed by exactly the same number of b's followed by exactly the same number of c's. This is an example of a context-sensitive language which is not also context-free; i.e., there is no context-free grammar for this language. An intuitive understanding of why this is true is beyond the scope of this text.

Sample Problem 3.0.2

Classify each of the following grammar rules according to Chomsky's classification of grammars (in each case give the largest - i.e. most restricted - classification type that applies):

1. $aSb \rightarrow aAcBb$

2. $B \rightarrow aA$

3. $S \rightarrow aBc$

4. $S \rightarrow aBc$

5. $Ab \rightarrow b$

6. $AB \rightarrow BA$

Solution:

1. *Type 1, Context-Sensitive*
2. *Type 3, Right Linear*
3. *Type 0, Unrestricted*
4. *Type 2, Context-Free*
5. *Type 1, Context-Sensitive*
6. *Type 0, Unrestricted*

3.0.3 Context-Free Grammars

Since programming languages are typically specified with context-free grammars, we are particularly interested in this class of grammars. Although there are some aspects of programming languages that cannot be specified with a context-free grammar, it is generally felt that using more complex grammars would only serve to confuse rather than clarify. In addition, context-sensitive grammars could not be used in a practical way to construct the compiler.

Context-free grammars can be represented in a form called Backus-Naur Form (BNF) in which nonterminals are enclosed in angle brackets <>, and the arrow is replaced by a ::=, as shown in the following example:

$< S >::= a < S > b$

which is the BNF version of the grammar rule:

$S \rightarrow aSb$

This form also permits multiple definitions of one nonterminal on one line, using the alternation vertical bar (|).

$< S >::= a < S > b|e$

which is the BNF version of two grammar rules:

$S \rightarrow aSb$

$S \rightarrow \epsilon$

BNF and context-free grammars are equivalent forms, and we choose to use context-free grammars only for the sake of appearance.

We now present some definitions which apply only to context-free grammars. A derivation tree is a tree in which each interior node corresponds to a nonterminal in a sentential form and each leaf node corresponds to a terminal symbol in the derived string. An example of a derivation tree for the string aaabbb, using grammar G2, is shown in Figure 3.2.

A context-free grammar is said to be *ambiguous* if there is more than one derivation tree for a particular string. In natural languages, ambiguous phrases

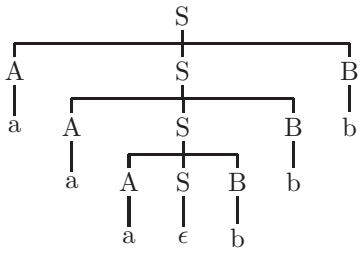


Figure 3.2: A derivation tree for aaabbb using grammar G2

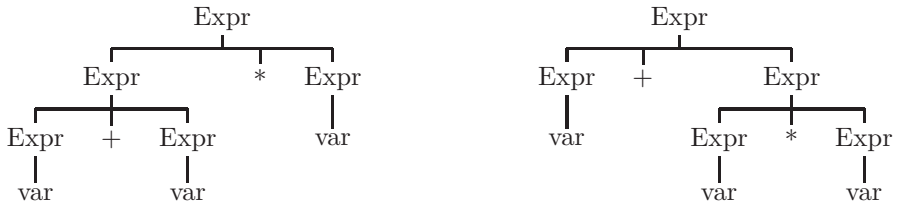


Figure 3.3: Two different derivation trees for the string var + var * var

are those which may have more than one interpretation. Thus, the derivation tree does more than show that a particular string is in the language of the grammar - it shows the structure of the string, which may affect the meaning or semantics of the string. For example, consider the following grammar for simple arithmetic expressions:

G4:

1. Expr → Expr + Expr
2. Expr → Expr * Expr
3. Expr → (Expr)
4. Expr → var
5. Expr → const

Figure 3.3 shows two different derivation trees for the string var+var*var, consequently this grammar is ambiguous. It should be clear that the second derivation tree in Figure 3.3 represents a preferable interpretation because it correctly shows the structure of the expression as defined in most programming languages (since multiplication takes precedence over addition). In other words, all subtrees in the derivation tree correspond to subexpressions in the derived expression. A nonambiguous grammar for expressions will be given in the next section.

A left-most derivation is one in which the left-most nonterminal is always the one to which a rule is applied. An example of a left-most derivation for grammar G2 above is:

$S \Rightarrow ASB \Rightarrow aSB \Rightarrow aASBB \Rightarrow aaSBB \Rightarrow aaBB \Rightarrow aabB \Rightarrow aabb$

We have a similar definition for right-most derivation. A left-most (or right-

most) derivation is a normal form for derivations; i.e., if two different derivations can be written in the same normal form, they are equivalent in that they correspond to the same derivation tree. Consequently, there is a one-to-one correspondence between derivation trees and left-most (or right-most) derivations for a grammar.

Sample Problem 3.0.3

Determine whether the following grammar is ambiguous. If so, show two different derivation trees for the same string of terminals, and show a left-most derivation corresponding to each tree.

1. $S \rightarrow aSbS$
2. $S \rightarrow aS$
3. $S \rightarrow c$

Solution:

$S \Rightarrow a\ S\ b\ S \Rightarrow a\ a\ S\ b\ S \Rightarrow a\ a\ c\ b\ S \Rightarrow a\ a\ c\ b\ c$
$S \Rightarrow a\ S \Rightarrow a\ a\ S\ b\ S \Rightarrow a\ a\ c\ b\ S \Rightarrow a\ a\ c\ b\ c$

We note that the two derivation trees correspond to two different left-most derivations, and the grammar is ambiguous.

3.0.4 Pushdown Machines

Like the finite state machine, the pushdown machine is another example of an abstract or theoretic machine. Pushdown machines can be used for syntax analysis, just as finite state machines are used for lexical analysis. A pushdown machine consists of:

- A finite set of states, one of which is designated the starting state.
- A finite set of input symbols, the input alphabet.
- An infinite stack and a finite set of stack symbols which may be pushed on top or removed from the top of the stack in a last-in first-out manner. The stack symbols need not be distinct from the input symbols. The stack must be initialized to contain at least one stack symbol before the first input symbol is read.
- A state transition function which takes as arguments the current state, the current input symbol, and the symbol currently on top of the stack; its result is the new state of the machine.
- On each state transition the machine may advance to the next input symbol or retain the input pointer (i.e., not advance to the next input symbol).
- On each state transition the machine may perform one of the stack operations, push(X) or pop, where X is one of the stack symbols.
- A state transition may include an exit from the machine labeled either Accept or Reject. This determines whether or not the input string is in the specified language.

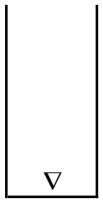
Note that without the infinite stack, the pushdown machine is nothing more than a finite state machine as defined in Chapter 2. Also, the pushdown machine halts by taking an exit from the machine, whereas the finite state machine halts when all input symbols have been read.

An example of a pushdown machine is shown in Figure 3.4, in which the rows are labeled by stack symbols and the columns are labeled by input symbols. The $\hookleftarrow$ character is used as an endmarker, indicating the end of the input string, and the $\bigtriangledown$ symbol is a stack symbol which we are using to mark the bottom of the stack so that we can test for the empty stack condition. The states of the machine are S1 (in our examples S1 will always be the starting state) and S2, and there is a separate transition table for each state. Each cell of those tables shows a stack operation (push() or pop), an input pointer function (advance or retain), and the next state. *Accept* and *Reject* are exits from the machine. The language of strings accepted by this machine is $\{a^n b^n\}$ where $n \geq 0$; i.e., the same language specified by grammar G2, above. To see this, the student should trace the operation of the machine for a particular input string. A trace showing the sequence of stack configurations and states of the machine for the input string aabb is shown in Figure 3.5. Note that while in state S1 the machine is pushing X's on the stack as each a is read, and while in state S2 the machine is popping an X off the stack as each b is read.

An example of a pushdown machine which accepts any string of correctly balanced parentheses is shown in Figure 3.6. In this machine, the input symbols are left and right parentheses, and the stack symbols are X and ,. Note that this language could not be accepted by a finite state machine because there could

S1	a	b	∇
X	Push (X) Advance S1	Pop Advance S2	Reject
↲	Push (X) Advance	Reject	Accept

S2	a	b	∇
X	Reject	Pop Advance S2	Reject
↲	Reject	Reject	Accept



Initial Stack

Figure 3.4: A pushdown machine to accept the language of grammar G2

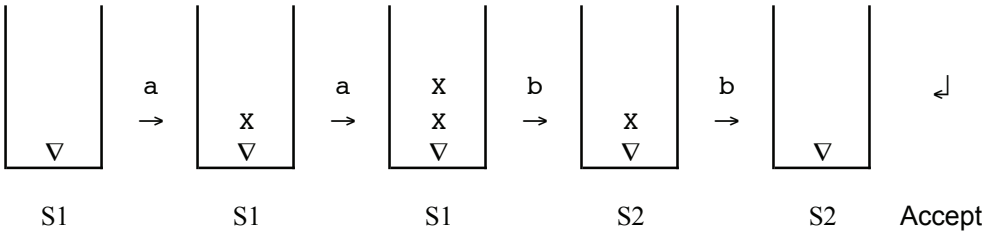


Figure 3.5: Sequence of stacks as the pushdown machine of Figure 3.4 accepts the input string aabb.

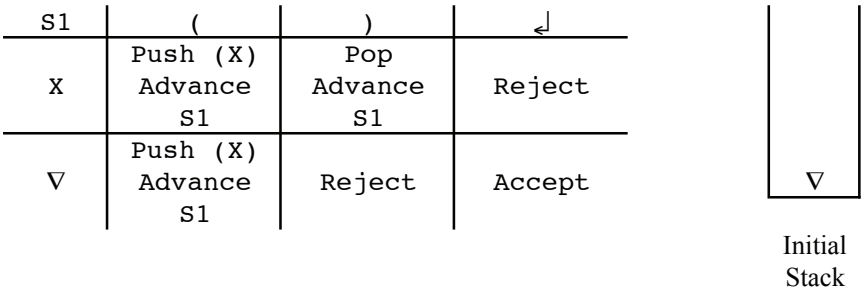


Figure 3.6: A pushdown machine to accept any string of well-balanced parentheses

be an unlimited number of left parentheses before the first right parenthesis. The student should compare the language accepted by this machine with the language of grammar G2.

The pushdown machines, as we have described them, are purely deterministic machines. A deterministic machine is one in which all operations are uniquely and completely specified regardless of the input (computers are deterministic), whereas a nondeterministic machine may be able to choose from zero or more operations in an unpredictable way. With nondeterministic pushdown machines it is possible to specify a larger class of languages. In this text we will not be concerned with nondeterministic machines.

We define a pushdown translator to be a machine which has an output function in addition to all the features of the pushdown machine described above. We may include this output function in any of the cells of the state transition table to indicate that the machine produces a particular output (e.g. Out(x)) before changing to the new state.

We now introduce an extension to pushdown machines which will make them easier to work with, but will not make them any more powerful. This extension is the Replace operation designated Rep(X,Y,Z,...), where X, Y, and Z are any stack symbols. The replace function replaces the top stack symbol with all the symbols in its argument list. The Replace function is equivalent to a pop operation followed by a push operation for each symbol in the argument list of the replace function. For example, the function `Rep (Term,+,Expr)` would pop the top stack symbol and push the symbols Term, +, and Expr in that order, as shown on the stack in Figure 3.7. (In this case, the stack symbols are separated by commas). Note that the symbols to be pushed on the stack are pushed in the order listed, left to right, in the Replace function. An extended pushdown machine is one which can use a Replace operation in addition to push and pop.

An extended pushdown machine is not capable of specifying any languages that cannot be specified with an ordinary pushdown machine; it is simply in-

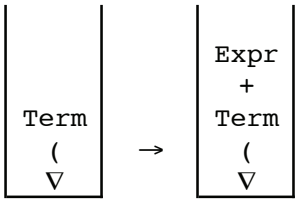


Figure 3.7: Effect on the stack of `Rep (Term, +, Expr)`

cluded here as a convenience to simplify some problem solutions. An extended pushdown translator is a pushdown translator which has a replace operation as defined above.

An example of an extended pushdown translator, which translates simple infix expressions involving addition and multiplication to postfix is shown in Figure 3.8, in which the input symbol a represents any variable or constant. An infix expression is one in which the operation is placed between the two operands, and a postfix expression is one in which the two operands precede the operation:

```
Infix                Postfix
2 + 3                2 3 +
2 + 3 * 5            2 3 5 * +
2 * 3 + 5            2 3 * 5 +
(2 + 3) * 5          2 3 + 5 *
```

Note that parentheses are never used in postfix notation. In Figure 3.8 the default state transition is to stay in the same state, and the default input pointer operation is advance. States S2 and S3 show only a few input symbols and stack symbols in their transition tables, because those are the only configurations which are possible in those states. The stack symbol E represents an expression, and the stack symbol L represents a left parenthesis. Similarly, the stack symbols E_p and L_p represent an expression and a left parenthesis on top of a plus symbol, respectively.

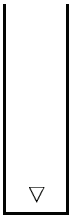
3.0.5 Correspondence Between Machines and Classes of Languages

We now examine the class of languages which can be specified by a particular machine. A language can be accepted by a finite state machine if, and only if, it can be specified with a right linear grammar (and if, and only if, it can be specified with a regular expression). This means that if we are given a right linear grammar, we can construct a finite state machine which accepts exactly the language of that grammar. It also means that if we are given a finite state

S1	a	+	*	(	)	↩
E	Reject	push(+)	push(*)	Reject	pop retain S3	pop retain
E_p	Reject	pop out(+)	push(*)	Reject	pop retain S2	pop retain S2
L	push(E) out(a)	Reject	Reject	push(L)	Reject	Reject
L_p	push(E) out(a)	Reject	Reject	push(L)	Reject	Reject
L_s	push(E) out(a)	Reject	Reject	push(L)	Reject	Reject
+	push(E_p) out(a)	Reject	Reject	push(L_p)	Reject	Reject
*	pop out(a*)	Reject	Reject	push(L_s)	Reject	Reject
▽	push(E) out(a)	Reject	Reject	push(L)	Reject	Accept

S2	)	↩
+	pop out(+) retain,S3	pop out(+) retain,S1
*	pop out(*) S1	Reject

S3	)
L	Rep(E) S1
L_p	Rep(E) S1
E	pop retain
L_s	pop retain S2
▽	Reject



Initial Stack

Figure 3.8: A pushdown translator for infix to postfix expressions

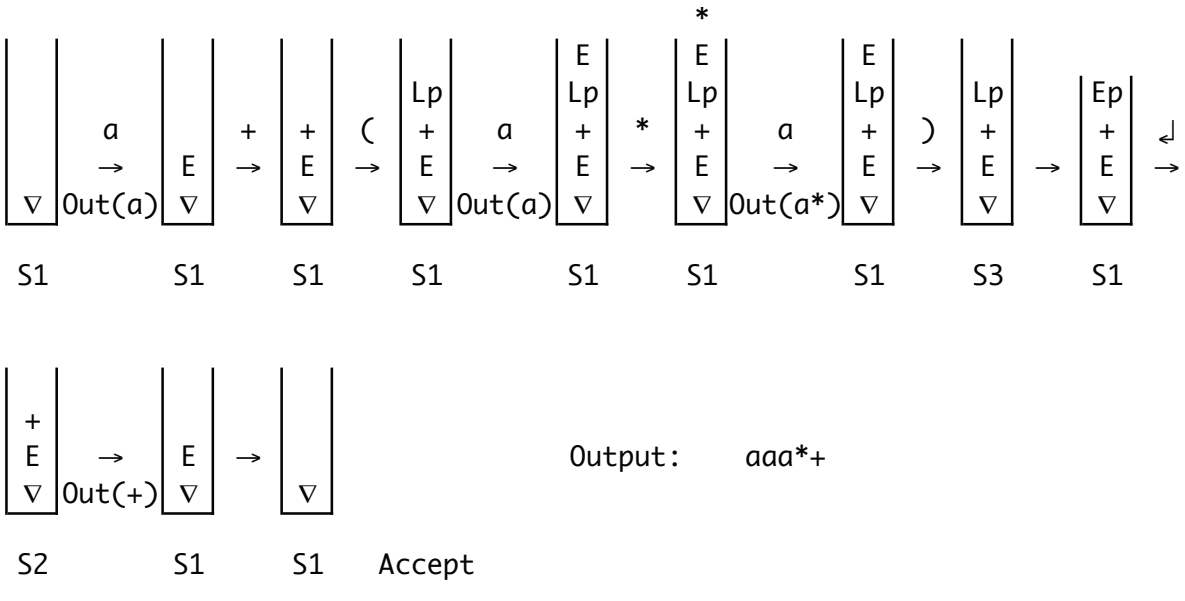
machine, we can write a right linear grammar which specifies the same language accepted by the finite state machine.

There are algorithms which can be used to produce any of these three forms (finite state machines, right linear grammars, and regular expressions), given one of the other two (see, for example, Hopcroft and Ullman [12]). However, here we rely on the student's ingenuity to solve these problems.

Sample Problem 3.0.4

Show the sequence of stacks and states which the pushdown machine of Figure 3.8 would go through if the input were: `a+(a*a)`

Solution:



Sample Problem 3.0.5

Give a right linear grammar for each of the languages of Sample Problem 2.0.2.

Solution:

(1) Strings over {0,1} containing an odd number of 0's.

1. $S \rightarrow 0$
2. $S \rightarrow 1S$
3. $S \rightarrow 0A$
4. $A \rightarrow 1$
5. $A \rightarrow 1A$
6. $A \rightarrow 0S$

(2) Strings over {0,1} which contain three consecutive 1's.

1. $S \rightarrow 1S$
2. $S \rightarrow 0S$
3. $S \rightarrow 1A$
4. $A \rightarrow 1B$
5. $B \rightarrow 1C$
6. $B \rightarrow 1$
7. $C \rightarrow 1C$
8. $C \rightarrow 0C$
9. $C \rightarrow 1$
10. $C \rightarrow 0$

(3) Strings over {0,1} which contain exactly three 0's.

1. $S \rightarrow 1S$
2. $S \rightarrow 0A$
3. $A \rightarrow 1A$
4. $A \rightarrow 0B$
5. $B \rightarrow 1B$
6. $B \rightarrow 0C$
7. $B \rightarrow 0$
8. $C \rightarrow 1C$
9. $C \rightarrow 1$

(4) Strings over {0,1} which contain an odd number of 0's and an even number of 1's.

1. $S \rightarrow 0A$
2. $S \rightarrow 1B$
3. $S \rightarrow 0$
4. $A \rightarrow 0S$
5. $A \rightarrow 1C$
6. $B \rightarrow 0C$
7. $B \rightarrow 1S$
8. $C \rightarrow 0B$
9. $C \rightarrow 1A$
10. $C \rightarrow 1$

We have a similar correspondence between machines and context-free languages. Any language which can be accepted by a deterministic pushdown machine can be specified by a context-free grammar. However, there are context-free languages which cannot be accepted by a deterministic pushdown machine. First consider the language, Pc, of palindromes over the alphabet {0,1} with centermarker, c. Pc = wcwr, where w is any string of 0's and 1's, and wr is w reversed. A grammar for Pc is shown below:
$S \rightarrow 0S0$
$S \rightarrow 1S1$
$S \rightarrow c$

Some examples of strings in this language are: c, 0c0, 110c011, 111c111.

The student should verify that there is a deterministic pushdown machine which will accept Pc. However, the language, P, of palindromes over the alphabet {0,1} without centermarker cannot be accepted by a deterministic pushdown machine. Such a machine would push symbols on the stack as they are input, but would never know when to start popping those symbols off the stack; i.e., without a centermarker it never knows for sure when it is processing the mirror image of the initial portion of the input string. For this language a nondeterministic pushdown machine, which is one that can pursue several different courses of action, would be needed. Nondeterministic machines are beyond the scope of this text. A grammar for P is shown below:

1. S $\rightarrow$ 0S0
2. S $\rightarrow$ 1S1
3. S $\rightarrow$ 0
4. S $\rightarrow$ 1
5. S $\rightarrow \epsilon$

The subclass of context-free languages which can be accepted by a deterministic pushdown machine are called deterministic context-free languages.

3.0.6 Exercises

1. Show three different *derivations* using each of the following grammars, with starting nonterminal S.

   (a)

   1. S $\rightarrow$ a S
   2. S $\rightarrow$ b A
   3. A $\rightarrow$ b S
   4. A $\rightarrow$ c

   (b)

   1. S $\rightarrow$ a B c
   2. B $\rightarrow$ A B
   3. A $\rightarrow$ B A
   4. A $\rightarrow$ a
   5. B $\rightarrow \epsilon$

   (c)

   1. S $\rightarrow$ a S B c
   2. a S A $\rightarrow$ a S b b
   3. B c $\rightarrow$ A c
   4. S b $\rightarrow$ b
   5. A $\rightarrow$ a

   (d)

   1. S $\rightarrow$ a b
   2. a $\rightarrow$ a A b B
   3. A b B $\rightarrow \epsilon$

2. Classify the grammars of the previous problem according to Chomsky's definitions (give the most restricted classification applicable).

3. Show an example of a *grammar rule* which is:

   ```
   (a)    Right Linear
   (b)    Context-Free, but not Right Linear
   (c)    Context-Sensitive, but not Context-Free
   (d)    Unrestricted, but not Context-Sensitive
   ```

4. For each of the given input strings show a derivation tree using the following grammar.

1. S → S a A
2. S → A
3. A → A b B
4. A → B
5. B → c S d
6. B → e
7. B → f

(a) eae (b) ebe (c) eaebe (d) ceaedbe (e) cebedaceaed

5. Show a left-most derivation for each of the following strings, using grammar G4 of section 3.0.3.

   (a) var + const (b) var + var * var (c) (var) (d) (var + var) * var

6. Show derivation trees which correspond to each of your solutions to the previous problem.

7. Some of the following grammars may be ambiguous; for each ambiguous grammar, show two different derivation trees for the same input string:

   (a)

   1. S → a S b
   2. S → A A
   3. A → c
   4. A → S

   (b)

   1. S → A a A
   2. S → A b A
   3. A → c
   4. A → z

   (c)

   1. S → a S b S
   2. S → a S
   3. S → c

   (d)

   1. S → a S b c
   2. S → A B
   3. A → a
   4. B → b

8. Show a pushdown machine that will accept each of the following languages:
(a) $\{a^n b^m\} m > n > 0$
(b) $a * (a + b)c*$
(c) $\{a^n b^n c^m d^m\} m, n \geq 0$
(d) $\{a^n b^m c^m d^n\} m, n > 0$
(e) $\{N_i c (N_{i+1})^r\}$
- where N_i is a binary representation of the integer i, and $(N_i)^r$ is N_i written right to left (reversed). Examples:

i	A string which should be accepted
19	10011c00101
19	10011c001010
15	1111c00001
15	1111c0000100

Hint: Use the first state to push N_i onto the stack until the c is read. Then use another state to pop the stack as long as the input is the complement of the stack symbol, until the top stack symbol and the input symbol are equal. Then use a third state to ensure that the remaining input symbols match the symbols on the stack. A fourth state can be used to allow for leading (actaully, trailing) zeros after the c.

9. Show the output and the sequence of stacks for the machine of Figure 3.8 for each of the following input strings:

(a) $a + a * a \hookleftarrow$
(b) $(a + a) * a \hookleftarrow$
(c) $(a) \hookleftarrow$
(d) $((a)) \hookleftarrow$

10. Show a grammar and an extended pushdown machine for the language of prefix expressions involving addition and multiplication. Use the terminal symbol a to represent a variable or constant. Example: *+aa*aa

11. Show a pushdown machine to accept palindromes over {0,1} with center-marker c. This is the language, Pc, referred to in section 3.0.5.

12. Show a grammar for the language of valid regular expressions (as defined in section 2.0) over the alphabet {0,1}. You may assume that concatenation is always represented by a raised dot. An example of a string in this language would be:
$(0 + 1 \cdot 1) * \cdot 0$
An example of a string not in this language would be:
$((0 + +1)$
Hint: Think about grammars for arithmetic expressions.

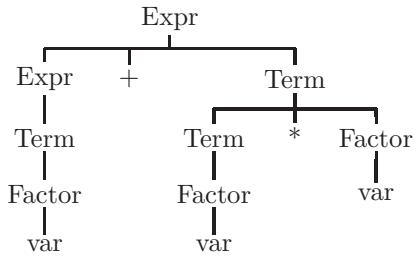


Figure 3.9: A derivation tree for `var + var * var` using grammar G5

3.1 Ambiguities in Programming Languages

Ambiguities in grammars for programming languages should be avoided. One way to resolve an ambiguity is to rewrite the grammar of the language so as to be unambiguous. For example, the grammar G4 in section 3.0.3 is a grammar for simple arithmetic expressions involving only addition and multiplication. As we observed, it is an ambiguous grammar because there exists an input string for which we can find more than one derivation tree. This ambiguity can be eliminated by writing an equivalent grammar which is not ambiguous:

G5:

1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor → (Expr)
6. Factor → var
7. Factor → const

A derivation tree for the input string `var + var * var` is shown, in Figure 3.9. The student should verify that there is no other derivation tree for this input string, and that the grammar is not ambiguous. Also note that in any derivation tree using this grammar, subtrees correspond to subexpressions, according to the usual precedence rules. The derivation tree in Figure 3.9 indicates that the multiplication takes precedence over the addition. The left associativity rule would also be observed in a derivation tree for `var + var + var`.

Another example of ambiguity in programming languages is the conditional statement as defined by grammar G6:

G6:

1. Stmt → IfStmt
2. IfStmt → if (BoolExpr) Stmt
3. IfStmt → if (BoolExpr) Stmt else Stmt

Think of grammar G6 as part of a larger grammar in which the nonterminal

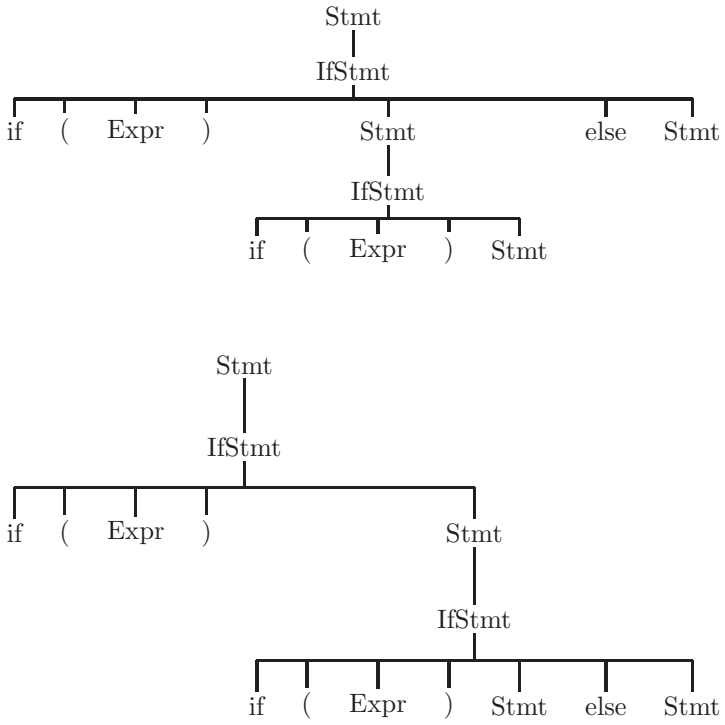


Figure 3.10: Two different derivation trees for the string `if ( Expr ) if ( Expr ) Stmt else Stmt`

Stmt is completely defined. For the present example we will show derivation trees in which some of the leaves are left as nonterminals. Two different derivation trees for the input string if (BoolExpr) if (BoolExpr) Stmt else Stmt are shown in Figure 3.10. In this grammar, a BoolExpr is any expression which results in a boolean (true/false) value. A Stmt is any statement, including if statements. This ambiguity is normally resolved by informing the programmer that elses always are associated with the closest previous unmatched ifs. Thus, the second derivation tree in Figure 3.10 corresponds to the correct interpretation. The grammar G6 can be rewritten with an equivalent grammar which is not ambiguous:

G7:

1. Stmt → IfStmt
2. IfStmt → Matched
3. IfStmt → Unmatched
4. Matched → if (BoolExpr) Matched else Matched
5. Matched → OtherStmt
6. Unmatched → if (BoolExpr) Stmt
7. Unmatched → if (BoolExpr) Matched else Unmatched

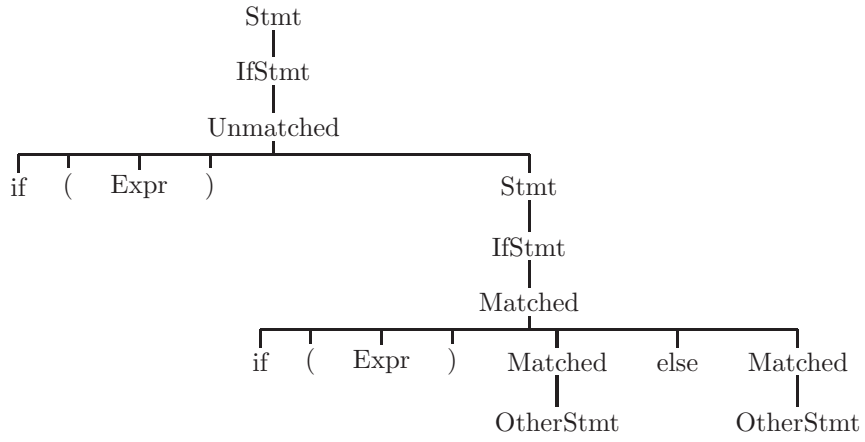


Figure 3.11: A derivation tree for the string `if ( Expr ) if ( Expr ) Stmt else Stmt` using grammar G7

This grammar differentiates between the two different kinds of *if* statements, those with a matching *else* (Matched) and those without a matching *else* (Unmatched). The nonterminal OtherStmt would be defined with rules for statements other than if statements (while, assignment, for, ...). A derivation tree for the string `if ( BoolExpr ) if ( BoolExpr ) OtherStmt else OtherStmt` is shown in Figure 3.11.

3.1.1 Exercises

1. Show derivation trees for each of the following input strings using grammar G5.

   (a) $var * var$

   (b) $(var * var) + var$

   (c) (var)

   (d) $var * var * var$

2. Extend grammar G5 to include subtraction and division so that subtrees of any derivation tree correspond to subexpressions.

3. Rewrite your grammar from the previous problem to include an exponentiation operator, `^`, such that `x^y` is x^y. Again, make sure that subtrees in a derivation tree correspond to subexpressions. Be careful, as exponentiation is usually defined to take precedence over multiplication and associate to the right:

   ```
   2*3^2 = 18  and  2^2^3 = 256
   ```

4. Two grammars are said to be isomorphic if there is a one-to-one correspondence between the two grammars for every symbol of every rule. For example, the following two grammars are seen to be isomorphic, simply by making the following substitutions: substitute B for A, x for a, and y for b.

   $S \rightarrow aAb$ $\quad$ $S \rightarrow xBy$
   $A \rightarrow bAa$ $\quad$ $B \rightarrow yBx$
   $A \rightarrow a$ $\quad$ $B \rightarrow x$

   Which *grammar* in section 3.0 is isomorphic to the grammar of Exercise 4 in section 3.1?

5. How many different derivation trees are there for each of the following `if` statements using grammar G6?

   (a) `if ( BoolExpr ) Stmt`

   (b) `if ( BoolExpr ) Stmt else if ( BoolExpr ) Stmt`

   (c) `if ( BoolExpr ) if ( BoolExpr ) Stmt else Stmt else Stmt`

   (d) `if ( BoolExpr ) if ( BoolExpr ) if ( BoolExpr ) Stmt else Stmt`

6. In the original C language it is possible to use assignment operators: `var =+ expr` means `var = var + expr` and `var =- expr` means `var = var - expr`. In later versions of C, C++, and Java the operator is placed before the equal sign:

   `var += expr` and `var -= expr`.

   Why was this change made?

3.2 The Parsing Problem

The student may recall, from high school days, the problem of diagramming English sentences. You would put words together into groups and assign syntactic types to them, such as noun phrase, predicate, and prepositional phrase. An example of a diagrammed English sentence is shown in Figure 3.12. The process of diagramming an English sentence corresponds to the problem a compiler must solve in the syntax analysis phase of compilation.

The syntax analysis phase of a compiler must be able to solve the parsing problem for the programming language being compiled: Given a grammar, G, and a string of input symbols, decide whether the string is in L(G); also, determine the structure of the input string. The solution to the parsing problem will be 'yes' or 'no', and, if 'yes', some description of the input string's structure, such as a derivation tree.

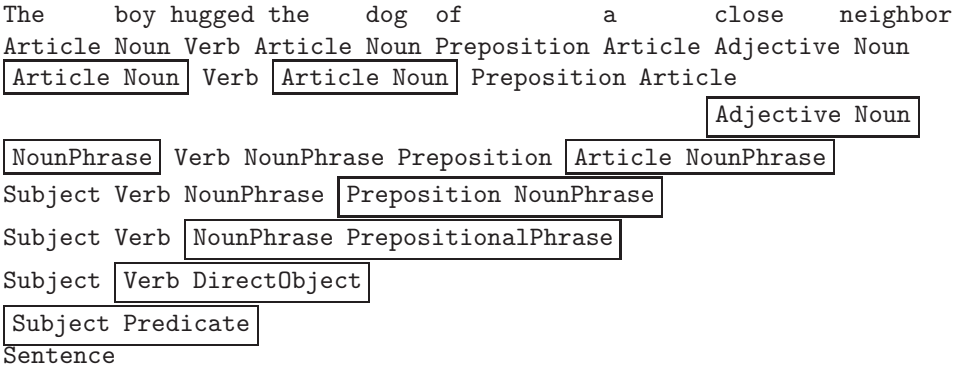


Figure 3.12: Diagram of an English sentence

A parsing algorithm is one which solves the parsing problem for a particular class of grammars. A good parsing algorithm will be applicable to a large class of grammars and will accommodate the kinds of rewriting rules normally found in grammars for programming languages. For context-free grammars, there are two kinds of parsing algorithms: bottom up and top down. These terms refer to the sequence in which the derivation tree of a correct input string is built. A parsing algorithm is needed in the syntax analysis phase of a compiler.

There are parsing algorithms which can be applied to any context-free grammar, employing a complete search strategy to find a parse of the input string. These algorithms are generally considered unacceptable since they are too slow; they cannot run in *polynomial time* (see Aho et. al. [1], for example).

3.3 Summary

This chapter on syntax analysis serves as an introduction to the chapters on parsing (chapters 4 and 5). In order to understand what is meant by parsing and how to use parsing algorithms, we first introduce some theoretic and linguistic concepts and definitions.

We define *grammar* as a finite list of *rewriting rules* involving *terminal* and *nonterminal* symbols, and we classify grammars in a hierarchy according to complexity. As we impose more restrictions on the rewriting rules of a grammar, we arrive at grammars for less complex languages. The four classifications of grammars (and languages) are (0) *unrestricted*, (1) *context-sensitive*, (2) *context-free*, and (3) *right linear*. The context-free grammars will be most useful in the syntax analysis phase of the compiler, since they are used to specify programming languages.

We define *derivations* and *derivation trees* for context-free grammars, which show the structure of a derived string. We also define *ambiguous grammars* as those which permit two different derivation trees for the same input string.

Pushdown machines are defined as machines having an infinite stack and are

shown to be the class of machines which corresponds to a subclass of context-free languages. We also define pushdown *translators* as pushdown machines with an output function, as this capability will be needed in compilers.

We take a careful look at ambiguities in programming languages, and see ways in which these ambiguities can be resolved. In particular, we look at grammars for simple arithmetic expressions and `if-else` statements.

Finally, we define the *parsing problem*: given a grammar and a string of input symbols, determine whether the string belongs to the language of the grammar, and, if so, determine its structure. We show that this problem corresponds exactly to the problem of diagramming an English sentence. The two major classifications of parsing algorithms are top-down, and bottom-up, corresponding to the sequence in which a derivation tree is built or traversed.

Chapter 4

Top Down Parsing

The parsing problem was defined in section 3.2 as follows: given a grammar and an input string, determine whether the string is in the language of the grammar, and, if so, determine its structure. Parsing algorithms are usually classified as either top down or bottom up, which refers to the sequence in which a derivation tree is built or traversed; in this chapter we consider only top down algorithms.

In a top down parsing algorithm, grammar rules are applied in a sequence which corresponds to a general top down direction in the derivation tree. For example, consider the grammar:

G8:

1. S $\rightarrow$ a S b
2. S $\rightarrow$ b A c
3. A $\rightarrow$ b S
4. A $\rightarrow$ a

We show a derivation tree for the input string `abbbaccb` in Figure 4.1. A parsing algorithm will read one input symbol at a time and try to decide, using the grammar, whether the input string can be derived. A top down algorithm will begin with the starting nonterminal and try to decide which rule of the grammar should be applied. In the example of Figure 4.1, the algorithm is able to make this decision by examining a single input symbol and comparing it with the first symbol on the right side of the rules. Figure 4.2 shows the sequence of events, as input symbols are read, in which the numbers in circles indicate which grammar rules are being applied, and the underscored symbols are the ones which have been read by the parser. Careful study of Figures 4.1 and 4.2 reveals that this sequence of events corresponds to a top down construction of the derivation tree.

In this chapter, we describe some top down parsing algorithms and, in addition, we show how they can be used to generate output in the form of atoms or syntax trees. This is known as syntax directed translation. However, we need to begin by describing the subclass of context-free grammars which can be parsed

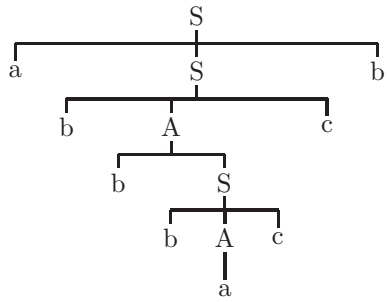


Figure 4.1: A derivation tree for `abbbaccb` using grammar G8

$S \Rightarrow \underline{a}Sb \Rightarrow \underline{ab}Acb \Rightarrow \underline{abb}Scb \Rightarrow \underline{abbb}Accb \Rightarrow \underline{abbbaccb}$
(1) (2) (3) (2) (4)

Figure 4.2: Sequence of events in a top down parse of the string `abbbaccb` using grammar G8

top down. In order to do this we begin with some preliminary definitions from discrete mathematics.

4.0 Relations and Closure

Whether working with top down or bottom up parsing algorithms, we will always be looking for ways to automate the process of producing a parser from the grammar for the source language. This will require, in most cases, the use of mathematics involving sets and relations. A relation is a set of ordered pairs. Each pair may be listed in parentheses and separated by commas, as in the following example:

R1:
(a,b)
(c,d)
(b,a)
(b,c)
(c,c)

Note that (a,b) and (b,a) are not the same. Sometimes the name of a relation is used to list the elements of the relation:
$4 < 9$
$5 < 22$
$2 < 3$
$-3 < 0$

If R is a relation, then the reflexive transitive closure of R is designated R*;

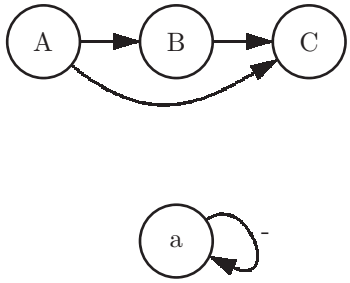


Figure 4.3: Transitive (top) and reflexive (bottom) aspects of a relation

it is a relation made up of the same elements of R with the following properties:

1. All pairs of R are also in R*.
2. If (a,b) and (b,c) are in R*, then (a,c) is in R* (Transitive).
3. If a is in one of the pairs of R, then (a,a) is in R* (Reflexive).

A diagram using an arrow to represent the relation can be used to show what we mean by *transitive* and *reflexive* properties and is shown in Figure 4.3. In rule 2 for transitivity note that we are searching the pairs of R*, not R. This means that as additional pairs are added to the closure for transitivity, those new pairs must also be checked to see whether they generate new pairs.

Sample Problem 4.0.1

Show R1 the reflexive transitive closure of the relation R1.*

Solution:

```
R1*:

(a,b)
(c,d)
(b,a)          (from R1)
(b,c)
(c,c)

(a,c)
```

```
(b,d)          (transitive)
(a,d)

(a,a)
(b,b)          (reflexive)
(d,d)
```

Note in Sample Problem 4.0.1 that we computed the transitive entries before the reflexive entries. The pairs can be listed in any order, but reflexive entries can never be used to derive new transitive pairs, consequently the reflexive pairs were listed last.

4.0.1 Exercises

1. Show the reflexive transitive closure of each of the following relations:

```
(a)     (a,b)          (b)     (a,a)          (c)     (a,b)
        (a,d)                  (a,b)                  (c,d)
        (b,c)                  (b,b)                  (b,c)
                                                      (d,a)
```

2. The mathematical relation *less than* is denoted by the symbol $<$. Some of the elements of this relation are: (4,5) (0,16) (-4,1) (1.001,1.002). What do we normally call the relation which is the *reflexive transitive closure* of *less than* ?

3. Write a program in Java or C++ to read in from the keyboard, ordered pairs (of strings, with a maximum of eight characters per string) representing a relation, and print out the reflexive transitive closure of that relation in the form of ordered pairs. You may assume that there will be, at most, 100 ordered pairs in the given relation, involving, at most, 100 different symbols.
   (Hint: Form a boolean matrix which indicates whether each symbol is related to each symbol).

4.1 Simple Grammars

At this point, we wish to show how top down parsers can be constructed for a given grammar. We begin by restricting the form of context-free grammar rules so that it will be very easy to construct a parser for the grammar. These gammars will not be very useful, but will serve as an appropriate introduction to top down parsing.

A grammar is a *simple* grammar if every rule is of the form:

$A \rightarrow a\alpha$

(where A represents any nonterminal, a represents any terminal, and α represents any string of terminals and nonterminals), and every pair of rules defining the same nonterminal begin with different terminals on the right side of the arrow. For example, the grammar G9 below is simple, but the grammar G10 is not simple because it contains an epsilon rule and the grammar G11 is not simple because two rules defining S begin with the same terminal.

G9:	G10:	G11:
$S \rightarrow aSb$	$S \rightarrow aSb$	$S \rightarrow aSb$
$S \rightarrow b$	$S \rightarrow \epsilon$	$S \rightarrow a$

A language which can be specified by a simple grammar is said to be a simple language. Simple languages will not be very useful in compiler design, but they serve as a good way of introducing the class of languages which can be parsed top down. The parsing algorithm must decide which rule of the grammar is to be applied as a string is parsed. The set of input symbols (i.e. terminal symbols) which imply the application of a grammar rule is called the *selection set* of that rule. For simple grammars, the selection set of each rule always contains exactly one terminal symbol - the one beginning the right hand side. In grammar G9, the selection set of the first rule is {a} and the selection set of the second rule is {b}. In top down parsing in general, rules defining the same nonterminal must have disjoint (non-intersecting) selection sets, so that it is always possible to decide which grammar rule is to be applied.

For example, consider grammar G12 below:

G12:

1. S → a b S d
2. S → b a S d
3. S → d

Figure 4.4 illustrates the construction of a derivation tree for the input string `abbaddd`, using grammar G12. The parser must decide which of the three rules to apply as input symbols are read. In Figure 4.4 the underscored input symbol is the one which determines which of the three rules is to be applied, and is thus used to guide the parser as it attempts to build a derivation tree. The input symbols which direct the parser to use a particular rule are the members of the selection set for that rule. In the case of simple grammars, there is exactly one symbol in the selection set for each rule, but for other context-free grammars, there could be several input symbols in the selection set.

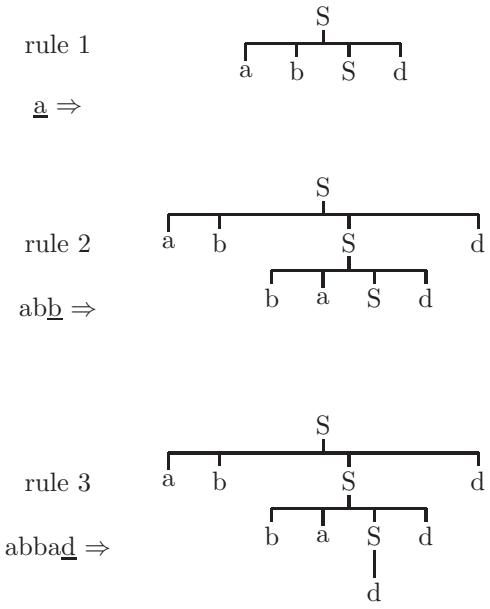


Figure 4.4: Using the input symbol to guide the parsing of the string `abbaddd`

4.1.1 Parsing Simple Languages with Pushdown Machines

In this section, we show that it is always possible to construct a one-state pushdown machine to parse the language of a simple grammar. Moreover, the construction of the machine follows directly from the grammar; i.e., it can be done mechanically. Consider the simple grammar G13 below:

G13:
1. S $\rightarrow$ aSB
2. S $\rightarrow$ b
3. B $\rightarrow$ a
4. B $\rightarrow$ bBa

We now wish to construct an extended pushdown machine to parse input strings consisting of a's and b's, according to the rules of this grammar. The strategy we use is to begin with a stack containing a bottom marker ($\bigtriangledown$) and the starting nonterminal, S. As the input string is being parsed, the stack will always correspond to the portion of the input string which has not been read. As each input symbol is read, the machine will attempt to apply one of the four rules in the grammar. If the top stack symbol is S, the machine will apply either rule 1 or 2 (since they define an S); whereas if the top stack symbol is B, the machine will apply either rule 3 or rule 4 (since they define a B). The current input symbol is used to determine which of the two rules to apply by comparing it with the selection sets (this is why we impose the restriction that

	a	b	↲
S	Rep (Bsa) Retain	Rep (b) Retain	Reject
B	Rep (a) Retain	Rep (aBb) Retain	Reject
a	pop Advance	Reject	Reject
b	Reject	pop Advance	Reject
∇	Reject	Reject	Accept

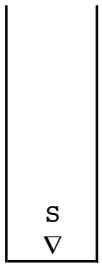


Figure 4.5: A pushdown machine for grammar G13

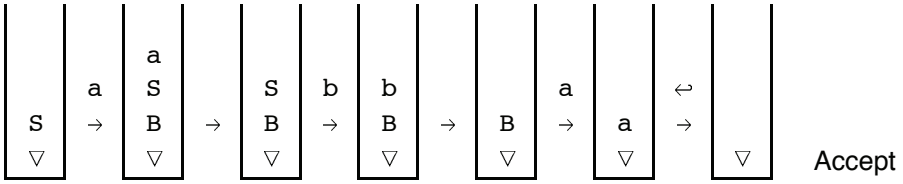


Figure 4.6: Sequence of stacks for machine of Figure 4.5 for input `aba`

rules defining the same nonterminal have disjoint selection sets).

Once we have decided which rule is to be entered in each cell of the pushdown machine table, it is applied by replacing the top stack symbol with the symbols on the right side of the rule in reverse order, and retaining the input pointer. This means that we will be pushing terminal symbols onto the stack in addition to nonterminals. When the top stack symbol is a terminal, all we need to do is ascertain that the current input symbol matches that stack symbol. If this is the case, simply pop the stack and advance the input pointer. Otherwise, the input string is to be rejected. When the end of the input string is encountered, the stack should be empty (except for $\bigtriangledown$) in order for the string to be accepted. The extended pushdown machine for grammar G13 is shown in Figure 4.5. The sequence of stack configurations produced by this machine for the input `aba` is shown in Figure 4.6.

In general, given a simple grammar, we can always construct a one state extended pushdown machine which will parse any input string. The construction of the machine could be done automatically:

1. Build a table with each column labeled by a terminal symbol (and end-marker $\hookrightarrow$) and each row labeled by a nonterminal or terminal symbol (and

bottom marker $\bigtriangledown$).

2. For each grammar rule of the form $A \rightarrow a\alpha$, fill in the cell in row A and column a with: $REP(a^r a), retain$, where a^r represents a reversed (here, a represents a terminal, and α represents a string of terminals and nonterminals).

3. Fill in the cell in row a and column a with pop, advance, for each terminal symbol a.

4. Fill in the cell in row $\bigtriangledown$, and column $\hookleftarrow$ with Accept.

5. Fill in all other cells with Reject.

6. Initialize the stack with $\bigtriangledown$, and the starting nonterminal.

This means that we could write a program which would accept, as input, any simple grammar and produce, as output, a pushdown machine which will accept any string in the language of that grammar and reject any string not in the language of that grammar. There is software which is able to do this for a grammar for a high level programming language; i.e., which is able to generate a parser for a compiler. Software which generates a compiler automatically from specifications of a programming language is called a *compiler-compiler*. We will study a popular compiler-compiler called *SableCC* in section 5.3.

4.1.2 Recursive Descent Parsers for Simple Grammars

A second way of implementing a parser for simple grammars is to use a methodology known as *recursive descent*, in which the parser is written using a traditional programming language, such as Java or C++. A method is written for each nonterminal in the grammar. The purpose of this method is to scan a portion of the input string until an *example* of that nonterminal has been read. By an example of a nontermninal, we mean a string of terminals or input symbols which can be derived from that nonterminal. This is done by using the first terminal symbol in each rule to decide which rule to apply. The method then handles each succeeding symbol in the rule; it handles nonterminals by calling the corresponding methods, and it handles terminals by reading another input symbol. For example, a recursive descent parser for grammar G13 is shown below:

```
class RDP                                  // Recursive Descent Parser
{
char inp;

public static void main (String[] args) throws IOException
{ InputStreamReader stdin = new InputStreamReader
                    (System.in);
   RDP rdp = new RDP();
       rdp.parse();
}

void parse ()
{ inp = getInp();
```

```
  S ();                              // Call start nonterminal
  if (inp=='N') accept();            // end of string marker
  else reject();
}

void S ()
{ if (inp=='a')                                    // apply rule 1
      {  inp = getInp();
         S ();
         B ();
       }                                           // end rule 1
  else if (inp=='b') inp = getInp();               // apply rule 2
  else reject();
}

void B ()
{ if (inp=='a') inp = getInp();                     // rule 3
  else if (inp=='b')                                // apply rule 4
       {  inp = getInp();
          B();
          if (inp=='a') inp = getInp();
          else reject();
       }                                           // end rule 4
   else reject();
}

void accept()                                       // Accept the input
{  System.out.println ("accept"); }

void reject()                                       // Reject the input
{  System.out.println ("reject");
   System.exit(0);                                  // terminate parser
}

char getInp()
{  try
     {  return (char) System.in.read(); }
   catch (IOException ioe)
     {  System.out.println ("IO error " + ioe); }
   return '#';                                 // must return a char
}
}
```

Note that the main method (parse) reads the first input character before calling the method for nonterminal S (the starting nonterminal). Each method

assumes that the initial input symbol in the example it is looking for has been read before that method has been called. It then uses the selection set to determine which of the grammar rules defining that nonterminal should be applied. The method S calls itself (because the nonterminal S is defined in terms of itself), hence the name *recursive* descent. When control is returned to the parse method, it must ensure that the entire input string has been read before accepting it. The methods accept() and reject() simply indicate whether or not the input string is in the language of the grammar. The method getInp() is used to obtain one character from the standard input file (keyboard). In subsequent examples, we omit the main, accept, reject, and getInp methods to focus attention on the concepts of recursive descent parsing. The student should perform careful hand simulations of this program for various input strings, to understand it fully.

Sample Problem 4.1.1

Show a one state pushdown machine and a recursive descent parser (show methods `S()` and `A()` only) for the following grammar:
1. $S \rightarrow 0S1A$
2. $S \rightarrow 10A$
3. $A \rightarrow 0S0$
4. $A \rightarrow 1$

Solution:

This grammar is simple because all rules begin with a terminal - rules 1 and 2 which define `S`, begin with different terminals, and rules 3 and 4 which define `A`, begin with different terminals.

The recursive descent parser is shown below:

```
void S()
   {  if (inp=='0')                // apply rule 1
         {  getInp();
            S();
            if (inp=='1') getInp();
            else Reject();
            A();
         }                         // end rule 1
      else if (inp=='1')           // apply rule 2
         {  getInpu();
            if (inp=='0') getInp();
            else reject();
            A();
         }                         // end rule 2
      else reject();
    }

void A()
   {  if (inp=='0')              // apply rule 3
         {  getInp();
            S();
            if (inp=='0') getInp();
            else reject();
         }                         // end rule 3
      else if (inp=='1') getInp()  // apply rule 4
      else reject();
   }
```

4.1.3 Exercises

1. Determine which of the following grammars are simple. For those which are simple, show an extended one-state pushdown machine to accept the language of that grammar.

   (a)

   1. S → a S b
   2. S → b

   (b)

   1. Expr → Expr + Term
   2. Expr → Term
   3. Term → var
   4. Term → (Expr)

   (c)

   1. S → a A b B
   2. A → b A
   3. A → a
   4. B → b A

   (d)

   1. S → a A b B
   2. A → b A
   3. A → b
   4. B → b A

   (e)

   1. S → a A b B
   2. A → b A
   3. A → ϵ
   4. B → b A

2. Show the sequence of stacks for the pushdown machine of Figure 4.5 for each of the following input strings:

   (a) aba↩

   (b) abbaa↩

   (c) aababaa↩

3. Show a recursive descent parser for each simple grammar of Problem 1, above.

4.2 Quasi-Simple Grammars

We now extend the class of grammars which can be parsed top down by permitting e rules in the grammar. A quasi-simple grammar is a grammar which obeys the restriction of simple grammars, but which may also contain rules of the form:

N $\rightarrow \epsilon$

(where N represents any nonterminal) as long as all rules defining the same nonterminal have disjoint selection sets. For example, the following is a quasi-simple grammar:

G14:

1. S $\rightarrow$ a A S
2. S $\rightarrow$ b
3. A $\rightarrow$ c A S
4. A $\rightarrow \epsilon$

In order to do a top down parse for this grammar, we will again have to find the selection set for each rule. In order to find the selection set for ϵ rules (such as rule 4) we first need to define some terms. The *follow set* of a nonterminal A, designated Fol(A), is the set of all terminals (or endmarker $\hookleftarrow$) which can immediately follow an A in an intermediate form derived from S $\hookleftarrow$, where S is the starting nonterminal. For grammar G14, above, the follow set of S is {a,b, $\hookleftarrow$} and the follow set of A is {a,b}, as shown by the following derivations:

$\underline{S \hookleftarrow} \Rightarrow aAS \hookleftarrow \Rightarrow acASS \hookleftarrow \Rightarrow acA\underline{Sa}AS \hookleftarrow$
$\Rightarrow acA\underline{Sb} \hookleftarrow$

Fol(S) = {a,b,$\hookleftarrow$}

$S \hookleftarrow \Rightarrow aAS \hookleftarrow \Rightarrow a\underline{Aa}AS \hookleftarrow$
$\Rightarrow a\underline{Ab} \hookleftarrow$

Fol(A) = {a,b}

For the present, we rely on the student's ingenuity to find all elements of the follow set. In a later section, we will present an algorithm for finding follow sets. The selection set for an ϵ rule is simply the follow set of the nonterminal on the left side of the arrow. For example, in grammar G14, above, the selection set of rule 4 is Sel(4) = Fol(A) = {a,b}. We use the follow set because these are the terminals which could be the current input symbol when, for instance, an example of an A in recursive descent is being sought.

To understand selection sets for quasi-simple grammars, consider the case where the parser for grammar G14 is attempting to build a derivation tree for the input string acbb. Once again, it is the selection set of each rule that guides the parser to apply that rule, as illustrated in Figure 4.7. If the parser is trying to decide how to rewrite an A, it will choose rule 3 if the input symbol is a c, but it will choose rule 4 if the input symbol is either an a or a b.

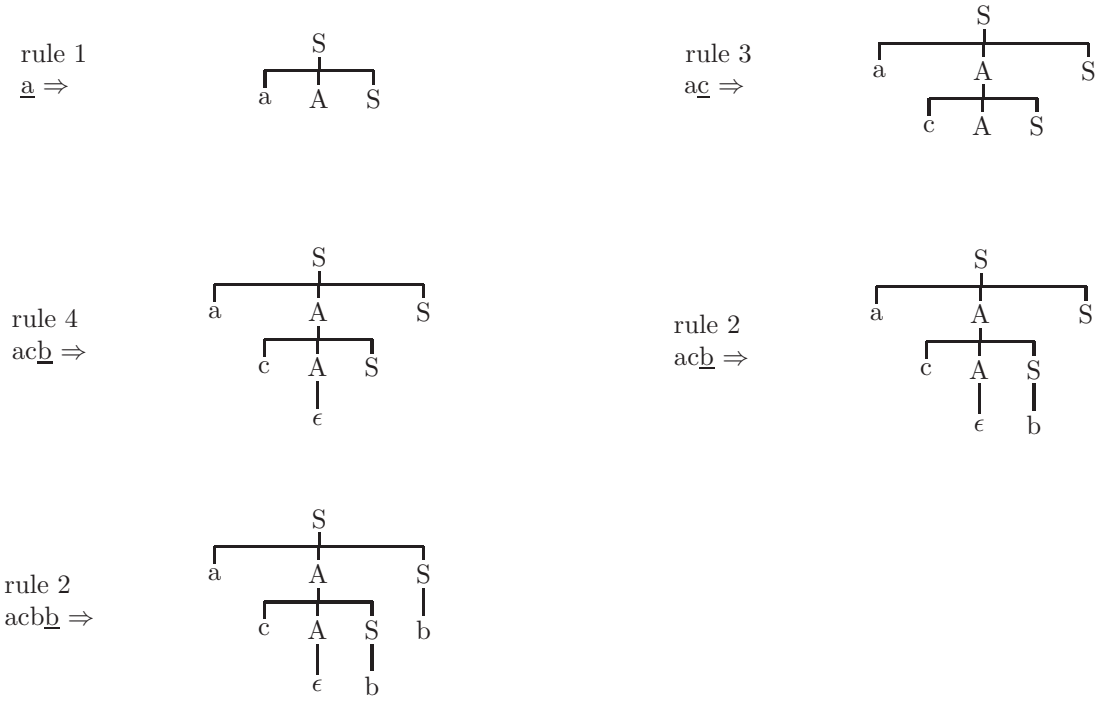


Figure 4.7: Construction of a Parse Tree for `acbb` using selection sets

	a	b	c	↲
S	Rep (SAa) Retain	Rep (b) Retain	Reject	Reject
A	Pop Retain	Pop Retain	Rep (SAc) Retain	Reject
a	Pop Advance	Reject	Reject	Reject
b	Reject	Pop Advance	Reject	Reject
c	Reject	Reject	Pop Advance	Reject
∇	Reject	Reject	Reject	Accept

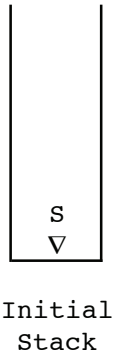


Figure 4.8: A pushdown machine for grammar G14

4.2.1 Pushdown Machines for Quasi-Simple Grammars

To build a pushdown machine for a quasi-simple grammar, we need to add only one step to the algorithm given in Section 4.1.1. We need to apply an ϵ rule by simply popping the nonterminal off the stack and retaining the input pointer. We do this only when the input symbol is in the follow set of the nonterminal defined in the ϵ rule. We would add the following step to the algorithm between steps 4 and 5:

4.5 For each ϵ rule in the grammar, fill in cells of the row corresponding to the nonterminal on the left side of the arrow, but only in those columns corresponding to elements of the follow set of the nonterminal. Fill in these cells with Pop, Retain. This will cause the machine to apply an ϵ rule by popping the nonterminal off the stack without reading any input symbols.

For example, the pushdown machine for grammar G14 is shown in Figure 4.8 Note, in particular, that the entries in columns a and b for row A (Pop, Retain) correspond to the e rule (rule 4).

4.2.2 Recursive Descent for Quasi-Simple Grammars

Recursive descent parsers for quasi-simple grammars are similar to those for simple grammars. The only difference is that we need to check for all the input symbols in the selection set of an ϵ rule. If any of these are the current input symbol, we simply return to the calling method without reading any input. By doing so, we are indicating that ϵ is an example of the nonterminal for which we are trying to find an example. A recursive descent parser for grammar G14 is shown below:

```
char inp;
void parse ()
     {     inp = getInp();
          S ();
          if (inp=='N') accept();
          else reject();
     }

void S ()
     {     if (inp=='a')                                  // apply rule 1
               {     inp = getInp();
                    A();
                    S();
               }
          // end rule 1
          else if (inp=='b') inp = getInp();     // apply rule 2
                 else reject();
     }

void A ()
     {     if (inp=='c')                                   // apply rule 3
               {     inp = getInp();
                    A ();
                    S ();
               }
          // end rule 3
          else if (inp=='a' || inp=='b') ;      // apply rule 4
                 else reject();
     }
```

Note that rule 4 is applied in method A() when the input symbol is a or b. Rule 4 is applied by returning to the calling method without reading any input characters. This is done by making use of the fact that Java permits null statements (at the comment // apply rule 4). It is not surprising that a null statement is used to process the null string.

4.2.3 A Final Remark on ϵ Rules

It is not strictly necessary to compute the selection set for ϵ rules in quasi-simple grammars. In other words, there is no need to distinguish between Reject entries and Pop, Retain entries in the row of the pushdown machine for an ϵ rule; they can all be marked Pop, Retain. If the machine does a Pop, Retain when it should Reject (i.e., it applies an ϵ rule when it really has no rule to apply), the syntax error will always be detected subsequently in the parse. However, this is often undesirable in compilers, because it is generally a good idea to detect syntax errors as soon as possible so that a meaningful error message can be put

out.

For example, in the pushdown machine of Figure 4.8, for the row labeled A, we have filled in Pop, Retain under the input symbols a and b, but Reject under the input symbol $\hookleftarrow$; the reason is that the selection set for the ϵ rule is {a,b}. If we had not computed the selection set, we could have filled in all three of these cells with Pop, Retain, and the machine would have produced the same end result for any input.

Sample Problem 4.2.1

Find the selection sets for the following grammar. Is the grammar quasi-simple? If so, show a pushdown machine and a recursive descent parser (show methods `S()` and `A()` only) corresponding to this grammar.

1. $S \rightarrow b\ A\ b$
2. $S \rightarrow a$
3. $A \rightarrow \epsilon$
4. $A \rightarrow a\ S\ a$

Solution:

In order to find the selection set for rule 3, we need to find the follow set of the nonterminal `A`. This is the set of terminals (including $\hookleftarrow$) which could follow an `A` in a derivation from $S \hookleftarrow$.

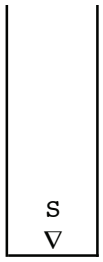
$S \hookleftarrow \Rightarrow b\underline{Ab}S \hookleftarrow$

The terminal `b` immediately follows an `A` in the derivation shown above. We cannot find any other teminals that can follow an `A` in a derivation from $S \hookleftarrow$. There fore, `FOL(A) =` {`b`}. The selection sets can now be listed: texttt Sel(1) = {b}

Sel(2) = {a}
Sel(3) = FOL(A) = {b}
Sel(4) = {a}

The grammar is quasi-simple because the rules defining an S have disjoint selection sets and the rules defining an A have dijoint selection sets. The pushdown machine is shown below:

	a	b	↲
S	Rep (a) Retain	Rep (bAb) Retain	Reject
A	Rep (aSa) Retain	Pop Retain	Reject
a	Pop Advance	Reject	Reject
b	Reject	Pop Advance	Reject
∇	Reject	Reject	Accept



Initial Stack

The recursive descent parser is shown below:

```
void S()
   {  if (inp=='b')                    // apply rule 1
         {  getInp();
            A();
            if (inp=='b') getInp();
            else Reject();
         }                             // end rule 1
      else if (inp=='a') getInp();     // apply rule 2
            else reject();
    }

void A()
   {  if (inp=='b')  ;                 // apply rule 3
      else if (inp=='a') getInp()      // apply rule 4
         {  getInp();
            S();
            if (inp=='a') getInp();
            else reject();
         }                            // end rule 4
      else reject();
   }
```

Note that rule 3 is implemented with a null statement. This should not be surprising since rule 3 defines A as the null string.

4.2.4 Exercises

1. Show the sequence of stacks for the pushdown machine of Figure 4.8 for each of the following input strings:

   (a) ab↩

   (b) acbb↩

   (c) aab↩

2. Show a derivation tree for each of the input strings in Problem 1, using grammar G14. Number the nodes of the tree to indicate the sequence in which they were applied by the pushdown machine.

3. Given the following grammar:

   1. S → a A b S
   2. S → ϵ
   3. A → a S b
   4. A → ϵ

   (a) Find the follow set for each nonterminal.

   (b) Show an extended pushdown machine for the language of this grammar.

   (c) Show a recursive descent parser for this grammar.

4.3 LL(1) Grammars

We now generalize the class of grammars that can be parsed top down by allowing rules of the form $N \rightarrow \alpha$ where α is any string of terminals and nonterminals. However, the grammar must be such that any two rules defining the same nonterminal must have disjoint selection sets. If it meets this condition, the grammar is said to be LL(1), and we can construct a one-state pushdown machine parser or a recursive descent parser for it. The name LL(1) is derived from the fact that the parser finds a left-most derivation when scanning the input from left to right if it can look ahead no more than one input symbol. In this section we present an algorithm to find selection sets for an arbitrary context-free grammar.

The algorithm for finding selection sets of any context-free grammar consists of twelve steps and is shown below. Intuitively, the selection set for each rule in the grammar is the set of terminals which the parser can expect to encounter when applying that grammar rule. For example, in grammar G15, below, we would expect the terminal symbol b to be in the selection set for rule 1, since:

$S \Rightarrow ABc \Rightarrow bABc$

In this discussion, the phrase *any string* always includes the null string, unless otherwise stated. As each step is explained, we present the result of that step when applied to the example, grammar G15.

G15:

1. S $\rightarrow$ ABc
2. A $\rightarrow$ bA
3. A $\rightarrow \epsilon$
4. B $\rightarrow$ c

Step 1. Find all nullable rules and nullable nonterminals:

Remove, temporarily, all rules containing a terminal. Allϵrules are nullable rules. The nonterminal defined in a nullable rule is a nullable nonterminal. In addition, all rules in the form

$A \rightarrow BCD...$

where B, C, D, ... are all nullable non-terminals, are nullable rules; the nonterminals defined by these rules are also nullable non-terminals. In other words, a nonterminal is nullable if ϵ can be derived from it, and a rule is nullable if ϵ can be derived from its right side.

For grammar G15:
Nullable rules: rule 3
Nullable nonterminals: A

Step 2. Compute the relation *Begins Directly With* for each nonterminal:

A BDW X if there is a rule $A \rightarrow \alpha X \beta$ such that α is a nullable string (a string of nullable nonterminals). A represents a nonterminal and X represents a terminal or nonterminal. β represents any string of terminals and nonterminals.

For G15:

```
S BDW A        (from rule 1)
S BDW B        (also from rule 1, because A is nullable)
A BDW b        (from rule 2)
B BDW c        (from rule 4)
```

Step 3. Compute the relation *Begins With*:

X BW Y if there is a string beginning with Y that can be derived from X. BW is the reflexive transitive closure of BDW. In addition, BW should contain pairs of the form a BW a for each terminal a in the grammar.

For G15:

```
S BW A
S BW B      (from BDW)
A BW b
B BW c

S BW b      (transitive)
S BW c
```

```
S BW S
A BW A
B BW B      (reflexive)
b BW b
c BW c
```

Step 4. Compute the set of terminals *First(x)* for each symbol x in the grammar.

At this point, we can find the set of all terminals which can begin a sentential form when starting with a given symbol of the grammar.

First(A) = set of all terminals b, such that A BW b for each nonterminal A. First(t) = {t} for each terminal t.

For G15:

```
First(S) = {b,c}
First(A) = {b}
First(B) = {c}
First(b) = {b}
First(c) = {c}
```

Step 5. Compute *First* of right side of each rule:

We now compute the set of terminals which can begin a sentential form derivable from the right side of each rule.

```
First (XYZ...) = First(X)
               U First(Y)   if X is nullable
               U First(Z)   if Y is also nullable
               .
               .
               .
```

In other words, find the union of the First(x) sets for each symbol on the right side of a rule, but stop when reaching a non-nullable symbol.

For G15:

```
1. First(ABc) = First(A) U First(B) = {b,c} (because A is nullable)
2. First(bA) =  {b}
3. First(e) =   {}
4. First(c) =   {c}
```

If the grammar contains no nullable rules, you may skip to step 12 at this point.

Step 6. Compute the relation *Is Followed Directly By*:

B FDB X if there is a rule of the form

$A \rightarrow \alpha B \beta X \gamma$

where β is a string of nullable nonterminals, α, γ are strings of symbols, X is any symbol, and A and B are nonterminals.

For G15:

```
A FDB B        (from rule 1)
B FDB c        (from rule 1)
```

Note that if B were a nullable nonterminal we would also have A FDB c.

Step 7. Compute the relation *Is Direct End Of*:

X DEO A if there is a rule of the form:

$A \rightarrow \alpha X \beta$

where β is a string of nullable nonterminals, α is a string of symbols, and X is a single grammar symbol.

For G15:

```
c DEO S         (from rule 1)
A DEO A         (from rule 2)
b DEO A         (from rule 2, since A is nullable)
c DEO B         (from rule 4)
```

Step 8. Compute the relation *Is End Of*:

X EO Y if there is a string derived from Y that ends with X. EO is the reflexive transitive closure of DEO. In addition, EO should contain pairs of the form N EO N for each nullable nonterminal, N, in the grammar.

For G15:

```
c EO S
A EO A          (from DEO)
b EO A
c EO B
                (no transitive entries)
c EO c
S EO S          (reflexive)
b EO b
B EO B
```

Step 9. Compute the relation *Is Followed By*:

W FB Z if there is a string derived from $S \hookleftarrow$ in which W is immediately followed by Z.

If there are symbols X and Y such that

```
W EO X
X FDB Y
Y BW Z
             then  W FB Z
```

For G15:

```
A EO A          A FDB B          B BW B                A FB B
                                 B BW c                A FB c
b EO A                           B BW B                b FB B
                                 B BW c                b FB c
B EO B          B FDB c          c BW c                B FB c
c EO B                           c BW c                c FB c
```

Step 10. Extend the FB relation to include endmarker:

A FB $\hookleftarrow$ if A EO S where A represents any nonterminal and S represents the starting nonterminal.

For G15:

S FB $\hookleftarrow$ because S EO S

There are now seven pairs in the FB relation for grammar G15.

Step 11. Compute the *Follow Set* for each nullable nonterminal:

The follow set of any nonterminal A is the set of all terminals, t, for which A FB t.

Fol(A) = {t: A FB t}

To find selection sets, we need find follow sets for nullable nonterminals only. For G15:

Fol(A) = {c} since A is the only nullable nonterminal and A FB c.

Step 12. Compute the *Selection Set* for each rule:

i. $A \rightarrow \alpha$

if rule i is not a nullable rule, then Sel(i) = First(α)
if rule i is a nullable rule, then Sel(i) = First(α) U Fol(A)

For G15:

Sel(1) = First(ABc) = {b,c}
Sel(2) = First(bA) = {b}
Sel(3) = First(ϵ) U Fol(A) = {} U {c} = {c}
Sel(4) = First(c) = {c}

1. Find nullable rules and nullable nonterminals.
2. Find Begins Directly With relation (BDW).
3. Find Begins With relation (BW).
4. Find First(x) for each symbol, x.
5. Find First(n) for the right side of each rule, n.
6. Find Followed Directly By relation (FDB).
7. Find Is Direct End Of relation (DEO).
8. Find Is End Of relation (EO).
9. Find Is Followed By relation (FB).
10. Extend FB to include endmarker.
11. Find Follow Set, Fol(A), for each nullable nonterminal, A.
12. Find Selection Set, Sel(n), for each rule, n.

Figure 4.9: Summary of algorithm to find selection sets of any context-free grammar

Notice that since we are looking for the follow set of a nullable nonterminal in step 12, we have actually done much more than necessary in step 9. In step 9 we need produce only those pairs of the form A FB t, where A is a nullable nonterminal and t is a terminal.

The algorithm is summarized in Figure 4.9. A context-free grammar is LL(1) if rules defining the same nonterminal always have disjoint selection sets. Grammer G15 is LL(1) because rules 2 and 3 (defining the nonterminal A) have disjoint selection sets (the selection sets for those rules have no terminal symbols in common). Note that if there are no nullable rules in the grammar, we can get the selection sets directly from step 5 – i.e., we can skip steps 6-11. A graph showing the dependence of any step in this algorithm on the results of other steps is shown in Figure 4.10. For example, this graph shows that the results of steps 3,6, and 8 are needed for step 9.

4.3.1 Pushdown Machines for LL(1) Grammars

Once the selection sets have been found, the construction of the pushdown machine is exactly as for quasi-simple grammars. For a rule in the grammar, $A \rightarrow a$, fill in the cells in the row for nonterminal A and in the columns for the selection set of that rule with Rep(a^r), Retain, where a^r represents the right side of the rule reversed. For ϵ rules, fill in Pop, Retain in the columns for the

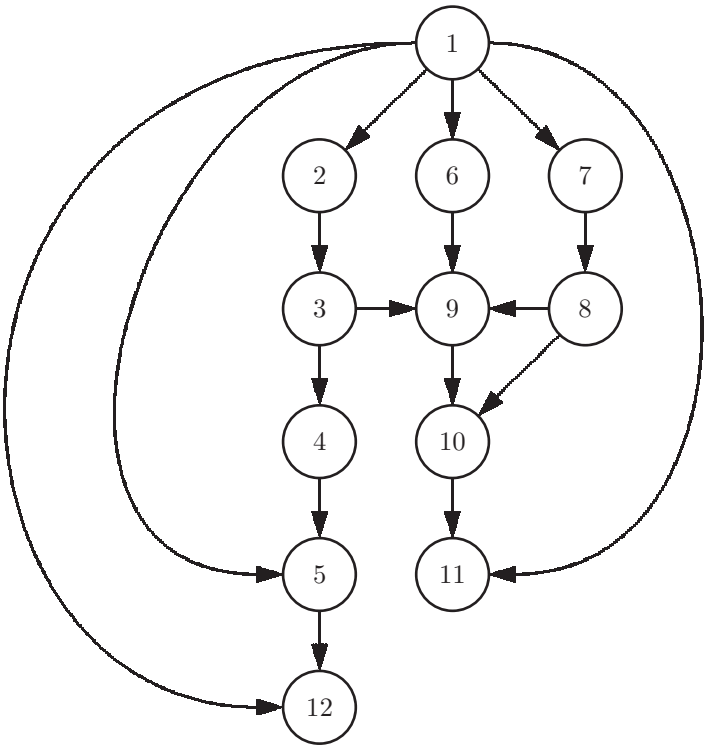
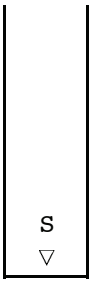


Figure 4.10: Dependency graph for the steps in the algorithm for finding selection sets

	b	c	↩
S	Rep (cBA) Retain	Rep (cBA) Retain	Reject
A	Rep (Ab) Retain	Pop Retain	Reject
B	Reject	Rep (c) Retain	Reject
b	Pop Advance	Reject	Reject
c	Reject	Pop Advance	Reject
▽	Reject	Reject	Accept



Initial Stack

Figure 4.11: A pushdown machine for grammar G15

selection set. For each terminal symbol, enter Pop, Advance in the cell in the row and column labeled with that terminal. The cell in the row labeled ▽ and the column labeled ↩ should contain Accept. All other cells are Reject. The pushdown machine for grammar G15 is shown in Figure 4.11.

4.3.2 Recursive Descent for LL(1) Grammars

Once the selection sets have been found, the construction of the recursive descent parser is exactly as for quasi-simple grammars. When implementing each rule of the grammar, check for the input symbols in the selection set for that grammar. A recursive descent parser for grammar G15 is shown below:

```
void parse ()
     {    getInp();
          S ();
          if (inp=='$\hookleftarrow$') accept; else reject();
     }

void S ()
     {     if (inp=='b' || inp=='c')              // apply rule 1
               {    A ();
                    B ();
                    if (inp=='c') getInp();
                    else reject();
               }                                  // end rule 1
          else reject();
     }
```

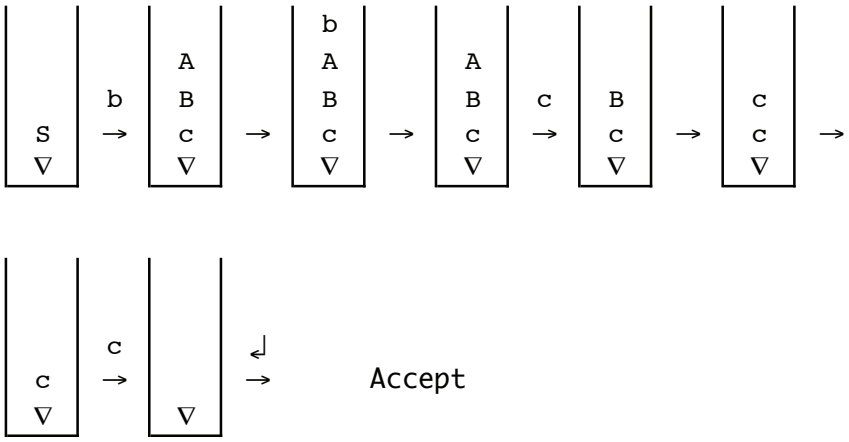
```
void A ()
     {     if (inp=='b')                          // apply rule 2
               {    getInp();
                    A ();
               }                                  // end rule 2
          else if (inp=='c') ;                    // apply rule 3
          else reject();
     }

void B ()
     {     if (inp=='c') getInp();               // apply rule 4
     else reject();
     }
```

Note that when processing rule 1, an input symbol is not read until a teminal is encountered in the grammar rule (after checking for a or b, an input symbol should not be read before calling procedure A).

Sample Problem 4.3.1

Show the sequence of stacks that occurs when the pushdown machine of Figure 4.11 parses the string `bcc`↩

Solution:



4.3.3 Exercises

1. Given the following information, find the Followed By relation (FB) as described in step 9 of the algorithm for finding selection sets:

```
A EO A     A FDB D     D BW b
A EO B     B FDB a     b BW b
B EO B                 a BW a
```

2. Find the selection sets of the following grammar and determine whether it is LL(1).
   1. $S \rightarrow ABD$
   2. $A \rightarrow aA$
   3. $A \rightarrow \epsilon$
   4. $B \rightarrow bB$
   5. $B \rightarrow \epsilon$
   6. $D \rightarrow dD$
   7. $D \rightarrow \epsilon$

3. Show a pushdown machine for the grammar of Problem 2.

4. Show a recursive descent parser for the grammar of Problem 2.

5. Step 3 of the algorithm for finding selection sets is to find the *Begins With* relation by forming the reflexive transitive closure of the *Begins Directly With* relation. Then add 'pairs of the form a BW a for each terminal a in the grammar'; i.e., there could be terminals in the grammar which do not

appear in the BDW relation. Find an example of a grammar in which the selection sets will not be found correctly if you do not add these pairs to the BW relation (hint: see step 9).

4.4 Parsing Arithmetic Expressions Top Down

Now that we understand how to determine whether a grammar can be parsed down, and how to construct a top down parser, we can begin to address the problem of building top down parsers for actual programming languages. One of the most heavily studied aspects of parsing programming languages deals with arithmetic expressions. Recall grammar G5 for arithmetic expressions involving only addition and multiplication, from Section 3.1. We wish to determine whether this grammar is LL(1).

G5:

1. $Expr \rightarrow Expr + Term$
2. $Expr \rightarrow Term$
3. $Term \rightarrow Term * Factor$
4. $Term \rightarrow Factor$
5. $Factor \rightarrow (Expr)$
6. $Factor \rightarrow var$

In order to determine whether this grammar is LL(1), we must first find the selection set for each rule in the grammar. We do this by using the twelve step algorithm given in Section 4.3.

```
1.      Nullable rules:        none
        Nullable nonterminals: none

2.      Expr              BDW       Expr
        Expr              BDW       Term
        Term              BDW       Term
        Term              BDW       Factor
        Factor            BDW       (
        Factor            BDW       var

3.      Expr              BW       Expr
        Expr              BW       Term
        Term              BW       Term
        Term              BW       Factor
        Factor            BW        (
        Factor            BW        var

        Factor            BW        Factor
        (                 BW       (
        var               BW       var
```

```
        Expr            BW      Factor
        Expr            BW      (
        Expr            BW      var
        Term            BW      (
        Term            BW      var
        *               BW      *
        +               BW      +
        )               BW      )

4.      First(Expr)          =  {(,var}
        First(Term)          =  {(,var}
        First(Factor)        =  {(,var}


5.      1.  First(Expr + Term)          = {(,var}
        2.  First(Term)                 = {(,var}
        3.  First(Term * Factor)        = {(,var}
        4.  First(Factor)               = {(,var}
        5.  First( ( Expr ) )           = {( }
        6.  First (var)                 = {var}

12.     Sel(1) = {(,var}
        Sel(2) = {(,var}
        Sel(3) = {(,var}
        Sel(4) = {(,var}
        Sel(5) = {( }
        Sel(6) = {var}
```

Since there are no nullable rules in the grammar, we can obtain the selection sets directly from step 5. This grammar is not LL(1) because rules 1 and 2 define the same nonterminal, Expr, and their selection sets intersect. This is also true for rules 3 and 4.

Incidentally, the fact that grammar G5 is not suitable for top down parsing can be determined much more easily by inspection of the grammar. Rules 1 and 3 both have a property known as left recursion:

1. $Expr \rightarrow Expr + Term$

3. $Term \rightarrow Term * Factor$

They are in the form:

$A \rightarrow Aa$

Note that any rule in this form cannot be parsed top down. To see this, consider the method for the nonterminal A in a recursive descent parser. The first thing it would do would be to call itself, thus producing infinite recursion with no base case (or 'escape hatch' from the recursion). Any grammar with left recursion cannot be LL(1).

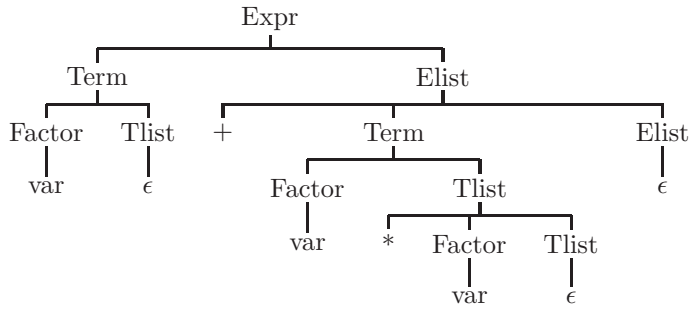


Figure 4.12: Derivation tree for the expression `var+var*var` using grammar G16

The left recursion can be eliminated by rewriting the grammar with an equivalent grammar that does not have left recursion. In general, the offending rule might be in the form:
$A \rightarrow A\alpha$
$A \rightarrow \beta$

in which we assume that β is a string of terminals and nonterminals that does not begin with an A. We can eliminate the left recursion by introducing a new nonterminal, R, and rewriting the rules as:
$A \rightarrow \beta R$
$R \rightarrow \alpha R$
$R \rightarrow \epsilon$

A more detailed and complete explanation of left recursion elimination can be found in Parsons [17].

This methodology is used to rewrite the grammar for simple arithmetic expressions in which the new nonterminals introduced are Elist and Tlist. An equivalent grammar for arithmetic expressions involving only addition and multiplication, G16, is shown below. A derivation tree for the expression `var+var*var` is shown in Figure 4.12.

G16:

1. Expr $\rightarrow$ Term Elist
2. Elist $\rightarrow$ + Term Elist
3. Elist $\rightarrow \epsilon$
4. Term $\rightarrow$ Factor Tlist
5. Tlist $\rightarrow$ * Factor Tlist
6. Tlist $\rightarrow \epsilon$
7. Factor $\rightarrow$ (Expr)
8. Factor $\rightarrow$ var

Note in grammar G16 that an Expr is still the sum of one or more Terms and a Term is still the product of one or more Factors, but the left recursion has

been eliminated from the grammar. We will see later that this grammar also defines the precedence of operators as desired. The student should construct several derivation trees using grammar G16 in order to be convinced that it is not ambiguous.

We now wish to determine whether this grammar is LL(1), using the algorithm to find selection sets:

```
1.     Nullable rules:          3,6
       Nullable nonterminals:   Elist, Tlist

2.    Expr        BDW  Term
      Elist       BDW  +
      Term        BDW  Factor
      Tlist       BDW  *
      Factor      BDW  (
      Factor      BDW  var

3.    Expr        BW     Term
      Elist       BW     +
      Term        BW     Factor            (from BDW)
      Tlist       BW     *
      Factor      BW     (
      Factor      BW     var

      Expr        BW     Factor
      Term        BW     (
      Term        BW     var                  (transitive)
      Expr        BW     (
      Expr        BW     var

      Expr        BW     Expr
      Term        BW     Term
      Factor      BW     Factor
      Elist       BW     Elist
      Tlist       BW     Tlist                  (reflexive)
      Factor      BW     Factor
      +           BW     +
      *           BW     *
      (           BW     (
      var         BW     var
      )           BW     )

4.    First (Expr)      =   {(,var}
      First (Elist)     =   {+}
      First (Term)      =   {(,var}
```

```
      First (Tlist)     =   {*}
      First (Factor)    =   {(,var}

5.    1.      First(Term Elist)        = {(,var}
      2.      First(+ Term Elist)      = {+}
      3.      First(epsilon)           = { }
      4.      First(Factor Tlist)      = {(,var}
      5.      First(* Factor Tlist)    = {*}
      6.      First(epsilon)           = { }
      7.      First(( Expr ))          = {( }
      8.      First(var)               = {var}

6.    Term        FDB  Elist
      Factor      FDB  Tlist
      Expr        FDB  )

7.    Elist       DEO      Expr
      Term        DEO      Expr
      Elist       DEO      Elist
      Term        DEO      Elist
      Tlist       DEO      Term
      Factor      DEO      Term
      Tlist       DEO      Tlist
      Factor      DEO      Tlist
      )           DEO      Factor
      var         DEO      Factor

8.    Elist       EO      Expr
      Term        EO      Expr
      Elist       EO      Elist
      Term        EO      Elist
      Tlist       EO      Term
      Factor      EO      Term                      (from DEO)
      Tlist       EO      Tlist
      Factor      EO      Tlist
      )           EO      Factor

      var         EO      Factor
      Tlist       EO      Expr
      Tlist       EO      Elist
      Factor      EO      Expr
      Factor      EO      Elist
      )           EO      Term
      )           EO      Tlist
      )           EO      Expr                      (transitive)
      )           EO      Elist
```

```
      var         EO      Term
      var         EO      Tlist
      var         EO      Expr
      var         EO      Elist

      Expr        EO      Expr
      Term        EO      Term
      Factor      EO      Factor
      )           EO      )
      var         EO      var                    (reflexive)
      +           EO      +
      *           EO       *
      (           EO      (
      Elist       EO      Elist
      Tlist       EO       Tlist

9.    Tlist     EO      Term      FDB    Elist     BW     +            Tlist FB +
                                                   BW      Elist
      Factor    EO                                 BW      +
                                                   BW      Elist
      var       EO                                 BW      +
                                                   BW      Elist
      Term      EO                                 BW      +
                                                   BW      Elist
      )         EO                                 BW      +
                                                   BW      Elist

      )         EO      Factor    FDB    Tlist     BW      *
                                                   BW      Tlist
      var       EO                                 BW      *
                                                   BW      Tlist
      Factor    EO                                 BW      *
                                                   BW      Tlist

      Elist     EO      Expr      FDB    )         BW      )            Elist FB )
      Tlist     EO      Expr                                            Tlist FB )
```

10.
Elist FB ↩
Term FB ↩
Expr FB ↩
Tlist FB ↩
Factor FB ↩
11.

```
Fol (Elist) = {),↩ }
Fol (Tlist) = {+,),↩ }
12.

Sel(1) = First(Term Elist) = {(,var}
Sel(2) = First(+ Term Elist) = { +}
Sel(3) = Fol(Elist) = {),↩}
Sel(4) = First(Factor Tlist) = {(,var}
Sel(5) = First(* Factor Tlist) = {*}
Sel(6) = Fol(Tlist) = {+,),↩}
Sel(7) = First( ( Expr ) ) = {(}
Sel(8) = First(var) = {var}
```

Since all rules defining the same nonterminal (rules 2 and 3, rules 5 and 6, rules 7 and 8) have disjoint selection sets, the grammar G16 is LL(1).

In step 9 we could have listed several more entries in the FB relation. For example, we could have listed pairs such as `var FB +` and `Tlist FB Elist.` These were not necessary, however; this is clear if one looks ahead to step 11, where we construct the follow sets for nullable nonterminals. This means we need to use only those pairs from step 9 which have a nullable nonterminal on the left and a terminal on the right. Thus, we will not need `var FB +` because the left member is not a nullable nonterminal, and we will not need `Tlist FB Elist` because the right member is not a terminal.

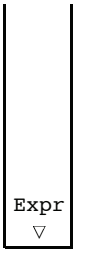
Sample Problem 4.4.1

Show an extended pushdown machine and a recursive descent parser for arithmetic expressions involving addition and multiplication using grammar G16.

Solution:

To build the pushdown machine we make use of the selection sets shown avove. These tell us which columns of the machine are to be filled in for each row. For example, since the selection set for rule 4 is {`(,var`}*, we fill the cells in the row labeled* `Term` *and columns labeled* `(` *and* `var` *with information from rule 4:* `Rep (Tlist Factor)`*. The solution is shown below.*

	+	*	(	)	var	↩
Expr	Reject	Reject	Rep(Elist Term) Retain	Reject	Rep(Elist Term) Retain	Reject
Elist	Rep(Elist Term +) Retain	Reject	Reject	Pop Retain	Reject	Pop Retain
Term	Reject	Reject	Rep(Tlist Factor) Retain	Reject	Rep(Tlist Factor) Retain	Reject
Tlist	Pop Retain	Rep(Tlist Factor *) Retain	Reject	Pop Retain	Reject	Pop Retain
Factor	Reject	Reject	Rep( )Expr( ) Retain	Reject	Rep(var) Retain	Reject
+	Pop Advance	Reject	Reject	Reject	Reject	Reject
*	Reject	Pop Advance	Reject	Reject	Reject	Reject
(	Reject	Reject	Pop Advance	Reject	Reject	Reject
)	Reject	Reject	Reject	Pop Advance	Reject	Reject
var	Reject	Reject	Reject	Reject	Pop Advance	Reject
▽	Reject	Reject	Reject	Reject	Reject	Accept



Initial Stack

We make use of the selection sets again in the recursive descent parser. In each procedure the input symbols in the selection set tell us which rule of the grammar to apply. Assume that a ***var*** *and the endmarker is each represented by an integer constant. The recursive descent parser is shown below:*

```
int inp;
final int var = 256;
final int endmarker = 257;

void Expr()
  {  if (inp=='(' || inp==var)            // apply rule 1
       {  Term();
```

```
          Elist();
        }                                   // end rule 1
      else reject();
  }

void Elist()
  {  if (inp=='+')                          // apply rule 2
       {  getInp();
          Term();
          Elist();
       }                                    // end rule 2
      else if (inp==')' || inp==endmarker)
          ;                                 // apply rule 3, null statement
      else reject();
  }

void Term()
  {  if (inp=='(' || inp==var)              // apply rule 4
       {  Factor();
          Tlist();
       }                                    // end rule 4
      else reject();
  }

void Tlist()
  {  if (inp=='*')                          // apply rule 5
       {  getInp();
          Factor();
          Tlist();
       }                                    // end rule 5
      else if (inp=='+' || inp==')'
              || inp==endmarker)
         ;                                  // apply rule 6, null statement
      else reject();
  }

void Factor()
  {  if (inp=='(')                          // apply rule 7
       {  getInp();
          Expr();
          if (inp==')') getInp();
          else reject();
       }                                   // end rule 7
      else if (inp==var) getInp();         // apply rule 8
      else reject();
  }
```

4.4.1 Exercises

1. Show derivation trees for each of the following input strings, using grammar G16.

   (a) var + var

   (b) var + var * var

   (c) (var + var) * var

   (d) ((var))

   (e) var * var * var

2. We have shown that grammar G16 for simple arithmetic expressions is LL(1), but grammar G5 is not LL(1). What are the advantages, if any, of grammar G5 over grammar G16?

3. Suppose we permitted our parser to *peek ahead* n characters in the input stream to determine which rule to apply. Would we then be able to use grammar G5 to parse simple arithmetic expressions top down? In other words, is grammar G5 LL(n)?

4. Find two null statements in the recursive descent parser of the sample problem in this section. Which methods are they in and which grammar rules do they represent?

5. Construct part of a recursive descent parser for the following portion of a programming language:

   1. $Stmt \rightarrow if(Expr)Stmt$
   2. $Stmt \rightarrow while(Expr)Stmt$
   3. $Stmt \rightarrow \{StmtList\}$
   4. $Stmt \rightarrow Expr;$

   Write the procedure for the nonterminal Stmt. Assume the selection set for rule 4 is {(, identifier, number}.

6. (a) Show an LL(1) grammar for the language of regular expressions over the alphabet {0,1}. Assume that concatenation is always designated by a raised dot.

(b) Show a derivation tree for the regular expression $1 + 0.1*$ (In the tree it should be clear that $1*$ is a subexpression)

(c) Show the selection set for each rule in your grammar.

(d) Show a recursive descent parser corresponding to the grammar.

(e) Show a one state pushdown machine corresponding to the grammar.

7. Show how to eliminate the left recursion from each of the grammars shown below:

(a) 1. $A \rightarrow Abc$
2. $A \rightarrow ab$

(b) 1. $ParmList \rightarrow ParmList, Parm$
2. $ParmList \rightarrow Parm$

8. A parameter list is a list of 0 or more parameters separated by commas; a parameter list neither begins nor ends with a comma. Show an LL(1) grammar for a parameter list. Assume that parameter has already been defined.

4.5 Syntax-Directed Translation

Thus far we have explored top down parsing in a way that has been exclusively concerned with syntax. In other words, the programs we have developed can check only for syntax errors; they cannot produce output, and they do not deal at all with semantics (the intent or meaning) of the source program. For this purpose, we now introduce action symbols which are intended to give us the capability of producing output and/or calling other methods while parsing an input string. A grammar containing action symbols is called a translation grammar. We will designate the action symbols in a grammar by enclosing them in curly braces {}. The meaning of the action symbol, by default, is to produce output: the action symbol itself.

To find the selection sets in a translation grammar, simply remove all the action symbols. This results in what is called the underlying grammar. Note that a rule of the form:

$A \rightarrow \{action\}$

is an epsilon rule in the underlying grammar. An example of a translation grammar to translate infix expressions involving addition and multiplication to postfix form is shown below:

G17:

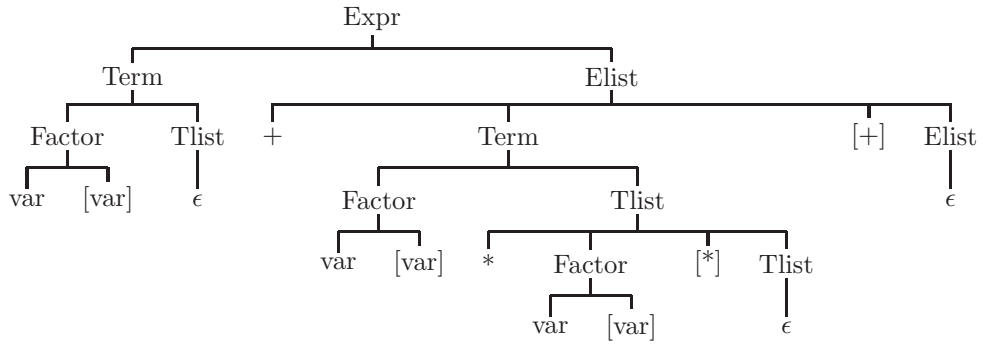


Figure 4.13: A derivation tree for the expression `var+var*var` using grammar G17

1. Expr → Term Elist
2. Elist → + Term {+} Elist
3. Elist → ϵ
4. Term → Factor Tlist
5. Tlist → * Factor {*} Tlist
6. Tlist → ϵ
7. Factor → (Expr)
8. Factor → var {var}

The underlying grammar of grammar G17 is grammar G16 in Section 4.4. A derivation tree for the expression var + var * var is shown in Figure 4.13. Note that the action symbols are shown as well. Listing the leaves of the derivation tree, the derived string is shown below:

var {var} + var {var} * var {var} {*} {+}

in which input symbols and action symbols are interspersed. Separating out the action symbols gives the output defined by the translation grammar:

{var} {var} {var} {*} {+}

4.5.1 Implementing Translation Grammars with Pushdown Translators

To implement a translation grammar with a pushdown machine, action symbols should be treated as stack symbols and are pushed onto the stack in exactly the same way as terminals and nonterminals occurring on the right side of a grammar rule. In addition, each action symbol {A} representing output should label a row of the pushdown machine table. Every column of that row should contain the entry Out(A), Pop, Retain. A pushdown machine to parse and translate infix expressions into postfix, according to the translation grammar G17, is shown in Figure 4.14, in which all empty cells are assumed to be Reject.

	var	+	*	(	)	↩
Expr	Rep(Elist Term) Retain			Rep(Elist Term) Retain		
Elist		Rep(Elist {+}Term+) Retain			Pop Retain	Pop Retain
Term	Rep(Tlist Factor) Retain			Rep(Tlist Factor) Retain		
Tlist		Pop Retain	Rep(Tlist {*}Factor*) Retain		Pop Retain	Pop Retain
Factor	Rep({var} var) Retain			Rep( )Expr( ) Retain		
var	Pop Advance					
+		Pop Advance				
*			Pop Advance			
(				Pop Advance		
)					Pop Advance	
{var}	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)	Pop Retain Out (var)
{+}	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)	Pop Retain Out (+)
{*}	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)	Pop Retain Out (*)
▽						Accept

Figure 4.14: An extended pushdown infix to postfix translator constructed from grammar G17

4.5.2 Implementing Translation Grammars with Recursive Descent

To implement a translation grammar with a recursive descent translator, action symbols should be handled as they are encountered in the right side of a grammar rule. Normally this will mean simply writing out the action symbol, but it could mean to call a method, depending on the purpose of the action symbol. The student should be careful in grammars such as G18:

G18:

1. S $\rightarrow$ {print}aS
2. S $\rightarrow$ bB
3. B $\rightarrow$ {print}

The method `S` for the recursive descent translator will print the action symbol *print* only if the input is an `a`. It is important always to check for the input symbols in the selection set before processing action symbols. Also, rule 3 is really an epsilon rule in the underlying grammar, since there are no terminals or nonterminals. Using the algorithm for selection sets, we find that:

`Sel(1) = {a}`
`Sel(2) = {b}`
`Sel(3) = {`$\hookleftarrow$`}`

The recursive descent translator for grammar G18 is shown below:

```
void S ()
{  if (inp=='a')
   {     getInp();               // apply rule 1
     System.out.println ("print");
     S();
   }                             // end rule 1
  else if (inp=='b')
  {       getInp();             // apply rule 2
     B();
  }                             // end rule 2
  else Reject ();
}

void B ()
{  if (inp==Endmarker) System.out.println ("print");
                                // apply rule 3
   else Reject ();
}
```

With these concepts in mind, we can now write a recursive descent translator to translate infix expressions to postfix according to grammar G17:

```
final int var = 256;                        // var token
char inp;
```

```
void Expr ()
{  if (inp=='(' || inp==var)
   {     Term ();                          // apply rule 1
     Elist ();
   }                                       // end rule 1
     else Reject ();
}

void Elist ()
{  if (inp=='+')
   {  getInp();                            // apply rule 2
     Term ();
     System.out.println ('+');
     Elist ();
   }                                      // end rule 2
   else if (inp==Endmarker || inp==')') ;     // apply rule 3
   else Reject ();
}

void Term ()
{  if (inp=='(' || inp==var)
   {  Factor ();                          // apply rule 4
     Tlist ();
   }                                      // end rule 4
     else Reject ();
}

void Tlist ()
{  if (inp=='*')
   {  getInp();                          // apply rule 5
     Factor ();
     System.out.println ('*');
     Tlist ();
   }                                    // end rule 5
     else if (inp=='+' || inp==')' || inp=Endmarker) ;
                                       // apply rule 6
     else Reject ();
}

void Factor ()
{  if (inp=='(')
   {  getInp();                       // apply rule 7
      Expr ();
      if (inp==')') getInp();
      else Reject ();
   }                                 // end rule 7
```

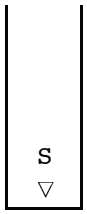
```
   else if (inp==var)
   {  getInp();                        // apply rule 8
      System.out.println ("var");
   }                                   // end rule 8
   else Reject ();
}
```

Sample Problem 4.5.1

Show an extended pushdown translator for the translation grammar G18.

Solution:

	a	b	↩
S	Rep (Sa{print}) Retain	Rep (Bv) Retain	Reject
B	Reject	Reject	Rep ({print}) Retain
a	Pop Adv		
b	Reject	Pop Adv	
{print}	Pop Retain Out ({print})	Pop Retain Out ({print})	Pop Retain Out ({print})
▽	Reject	Reject	Accept



Initial Stack

4.5.3 Exercises

1. Consider the following translation grammar with starting nonterminal S, in which action symbols are put out:

   1. S → A b B
   2. A → {w} a c
   3. A → b A {x}
   4. B → {y}

   Show a derivation tree and the output string for the input bacb.

2. Show an extended pushdown translator for the translation grammar of Problem 1.

3. Show a recursive descent translator for the translation grammar of Problem 1.

4. Write the Java statement which would appear in a recursive descent parser for each of the following translation grammar rules:

   (a) $A \rightarrow \{w\}a\{x\}BC$

   (b) $A \rightarrow a\{w\}\{x\}BC$

   (c) $A \rightarrow a\{w\}B\{x\}C$

4.6 Attributed Grammars

It will soon become clear that many programming language constructs cannot be adequately described with a context-free grammar. For example, many languages stipulate that a loop control variable must not be altered within the scope of the loop. This is not possible to describe in a practical way with a context-free grammar. Also, it will be necessary to propagate information, such as a location for the temporary result of a subexpression, from one grammar rule to another. Therefore, we extend grammars further by introducing attributed grammars, in which each of the terminals and nonterminals may have zero or more attributes, normally designated by subscripts, associated with it. For example, an attribute on an Expr nonterminal could be a pointer to the stack location containing the result value of an evaluated expression. As another example, the attribute of an input symbol (a lexical token) could be the value part of the token.

In an attributed grammar there may be zero or more attribute computation rules associated with each grammar rule. These attribute computation rules show how the attributes are assigned values as the corresponding grammar rule

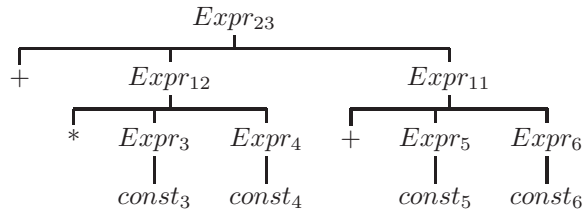


Figure 4.15: A attrubuted derivation tree for the prefix expression `+ * 3 4 + 5 6` using grammar G19

is applied during the parse. One way for the student to understand attributed grammars is to build derivation trees for attributed grammars. This is done by first eliminating all attributes from the grammar and building a derivation tree. Then, attribute values are entered in the tree according to the attribute computation rules. Some attributes take their values from attributes of higher nodes in the tree, and some atttributes take their values from attributes of lower nodes in the tree. For this reason, the process of filling in attribute values is not straightforward.

As an example, grammar G19 is an attributed (simple) grammar for prefix expressions involving addition and multiplication. The attributes, shown as subscripts, are intended to evaluate the arithmetic expression. The attribute on the terminal const is the value of the constant as supplied by the lexical phase. Note that this kind of expression evaluation is typical of what is done in an interpreter, but not in a compiler.

G19:

1. $Expr_p \rightarrow +Expr_qExpr_r$ p $\leftarrow$ q + r
2. $Expr_p \rightarrow *Expr_qExpr_r$ p $\leftarrow$ q * r
3. $Expr_p \rightarrow const_q$ p $\leftarrow$ q

An attributed derivation tree for the input `+ * 3 4 + 5 6` is shown in Figure 4.15. The attribute on each subexpression is the value of that subexpression. This example was easy because all of the attribute values are taken from a lower node in the tree. These are called *synthesized attributes*.

A second example, involving attribute values taken from higher nodes in the tree is shown in Figure 4.16. These are called *inherited attributes*. As an example of a grammar with inherited attributes, the following is a grammar for declarations of a numeric data type. In grammar G20, `type` is a terminal (token) whose value part may be a type such as int, float, etc. This grammar is used to specify type declarations, such as `int x,y,z;`. We wish to store the type of each identifier with its symbol table entry. To do this, the type must be passed down the derivation tree to each of the variables being declared.

G20:

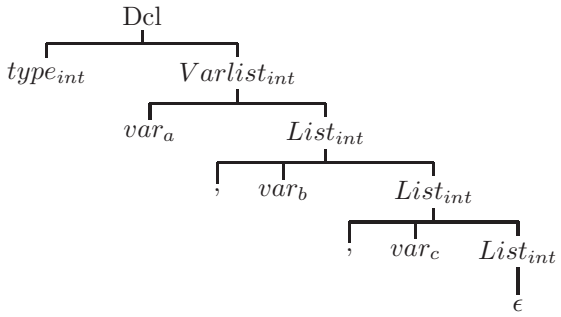


Figure 4.16: A attrubuted derivation tree for `int a,b,c;` using grammar G20

1. $Dcl \rightarrow type_t Varlist_w;$ w ← t
2. $VarList_w \rightarrow var_x List_y$ y ← w
3. $List_w \rightarrow ,var_x List_y$ y ← w
4. $List_w \rightarrow \epsilon$

An attributed derivation tree for the input string `int a,b,c` is shown in Figure 4.16. Note that the values of the attributes move either horizontally on one level (rule 1) or down to a lower level (rules 2 and 3). It is important to remember that the number and kind of attributes of a symbol must be consistent throughout the entire grammar. For example, if the symbol $A_{i,s}$ has two attributes, the first being inherited and the second synthesized, then this must be true everywhere the symbol A appears in the grammar.

4.6.1 Implementing Attributed Grammars with Recursive Descent

To implement an attributed grammar with recursive descent, the attributes will be implemented as parameters or instance variables in the methods defining nonterminals. For example, if $S_{a,b}$ is a nonterminal with two attributes, then the method S will have two parameters, a and b. Synthesized attributes are used to return information to the calling method and, hence, must be implemented with objects (i.e. with reference types). If the attribute is to store a whole number, we would be tempted to use the Java wrapper class Integer for synthesized attributes. Unfortunately, the Integer class is not mutable, i.e. Integer objects cannot be changed. Thus we will build our own wrapper class, called MutableInt, to store a whole number whose value can be changed. This class is shown in below:

```
// Wrapper class for ints which lets you change the value.
// This class is needed to implement attributed grammars
// with recursive descent

class MutableInt extends Object
```

```
{ int value;            // store a single int

MutableInt (int i)      // Initializing constructor
{  value = i; }

MutableInt ()            // Default constructor
{  value = 0;           // default value is 0
}

int get()                  // Accessor
{ return value; }

void set (int i)            // Mutator
{ value = i; }

public String toString()      // For printing
{  return "" + value; }
}
```

Since inherited attributes pass information to the called method, they may be passed by value or by using primitive types. Action symbol attributes will be implemented as instance variables. Care must be taken that the attribute computation rules are included at the appropriate places. Also, care must be taken in designing the attributed grammar that the computation rules do not constitute a contradiction. For example:

$S_p \leftarrow aA_rB_s \qquad p \leftarrow r + s$

The recursive descent method for S would be:

```
void S (MutableInt p)
{      if (token.getClass()=='a')
    {  token.getToken();
       A(r);
       B(s);
    // this must come after calls to  A(r), B(s)
       p.set(r.get() + s.get());
    }
}
```

In this example, the methods S, A, and B (A and B are not shown) all return values in their parameters (they are synthesized attributes, implemented with references), and there is no contradiction. However, assuming that the attribute of the B is synthesized, and the attribute of the A is inherited, the following rule could not be implemented:

$S \rightarrow aA_pB_q \qquad p \leftarrow q$

In the method S, q will not have a value until method B has been called and terminated. Therefore, it will not be possible to assign a value to p before calling method A. This assumes, as always, that input is read from left to right.

Sample Problem 4.6.1

Show a recursive descent parser for the attributed grammar G19. Assume that the `Token` *class has accessor methods,* `getClass()` *and* `getVal()`*, which return the class and value parts of a lexical token, respectively. The method* `getToken()` *reads in a new token.*

Solution:

```
   class RecDescent
{
// int codes for token classes
final int Num=0; // numeric constant
final int Op=1;                 // operator
final int End=2;                // endmarker
Token token;

void Eval ()
{  MutableInt p = new MutableInt(0);
   token = new Token();
   token.getToken();            // Read a token from stdin
   Expr(p);
                                // Show final result
   if (token.getClass() == End)
     System.out.println (p);
   else reject();
}

void Expr (MutableInt p)
{  MutableInt q = new MutableInt(0),     // Attibutes q,r
               r = new MutableInt(0);
   if (token.getClass()==Op)
     if (token.getVal()==(int)'+')       // apply rule 1
       {  token.getToken();              // read next token
          Expr(q);
          Expr(r);
          p.set (q.get() + r.get());
```

```
        }                                   // end rule 1
      else                                  // should be a *, apply rule 2
        {  token.getToken();                // read next token
           Expr(q);
           Expr(r);
           p.set (q.get() * r.get());
        }                                   // end rule 2
    else if (token.getClass() == Num)       // is it a numeric constant?
      {  p.set (token.getVal());            // apply rule 3
         token.getToken();                  // read next token
      }                                     // end rule 3
    else reject();
}
```

4.6.2 Exercises

1. Consider the following attributed translation grammar with starting nonterminal S, in which action symbols are output:

   1. $S_p \rightarrow A_q b_r A_t \qquad p \leftarrow r + t$
   2. $A_p \rightarrow a_p \{w\}_p c$
   3. $A_p \rightarrow b_q A_r \{x\}_p \qquad p \leftarrow q + r$

   Show an attributed derivation tree for the input string $a_1 c b_2 b_3 a_4 c$, and show the output symbols with attributes corresponding to this input.

2. Show a recursive descent translator for the grammar of Problem 1. Assume that all attributes are integers and that, as in sample problem 4.6, the Token class has methods getClass() and getValue() which return the class and value parts of a lexical token, and the Token class has a getToken() method which reads a token from the standard input file. p

3. Show an attributed derivation tree for each input string using the following attributed grammar:

   1. $S_p \rightarrow A_{q,r} B_t \qquad p \leftarrow q * t$
      $\qquad\qquad\qquad\qquad\quad r \leftarrow q + t$
   2. $A_{p,q} \rightarrow b_r A_{t,u} c \qquad u \leftarrow r$

	$p \leftarrow r + t + u$
3. $A_{p,q} \rightarrow \epsilon$	$p \leftarrow 0$
4. $B_p \rightarrow a_p$	

(a) a_2

(b) $b_1 c a_3$

(c) $b_2 b_3 c c a_4$

4. Is it possible to write a recursive descent parser for the attributed translation grammar of Problem 3?

4.7 An Attributed Translation Grammar for Expressions

In this section, we make use of the material presented thus far on top down parsing to implement a translator for infix expressions involving addition and multiplication. The output of the translator will be a stream of atoms, which could be easily translated to the appropriate instructions on a typical target machine. Each atom will consist of four parts: (1) an operation, ADD or MULT, (2) a left operand, (3) a right operand, and (4) a result. (Later, in section 4.8, we will need to add two more fields to the atoms.) For example, if the input were A + B * C + D, the output could be the following three atoms:

```
MULT        B       C       T1
ADD         A       T1      T2
ADD         T2      D       T3
```

Note that our translator will have to find temporary storage locations (or use a stack) to store intermediate results at run time. It would indicate that the final result of the expression is in T3. In the attributed translation grammar G21, shown below, all nonterminal attributes are synthesized, with the exception of the first attribute on Elist and Tlist, which are inherited:

G21:

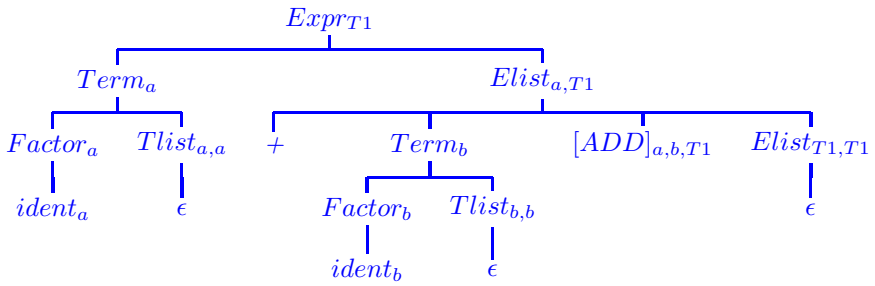
1. $Expr_p$	$\rightarrow Term_q Elist_{q,p}$	
2. $Elist_{p,q}$	$\rightarrow +Term_r \{ADD\}_{p,r,s} Elist_{s,q}$	$s \leftarrow alloc()$
3. $Elist_{p,q}$	$\rightarrow \epsilon$	$q \leftarrow p$
4. $Term_p$	$\rightarrow Factor_q Tlist_{q,p}$	
5. $Tlist_{p,q}$	$\rightarrow *Factor_r \{MULT\}_{p,r,s} Tlist_{s,q}$	$s \leftarrow alloc()$
6. $Tlist_{p,q}$	$\rightarrow \epsilon$	$q \leftarrow p$
7. $Factor_p$	$\rightarrow (Expr_p)$	
8. $Factor_p$	$\rightarrow ident_p$	

The intent of the action symbol $\{ADD\}_{p,r,s}$ is to put out an ADD atom with operands p and r and result s. In many rules, several symbols share an attribute; this means that the attribute is to have the same value on those symbols. For example, in rule 1 the attribute of Term is supposed to have the same value as the first attribute of Elist. Consequently, those attributes are given the same name. This also could have been done in rules 3 and 6, but we chose not to do so in order to clarify the recursive descent parser. For this reason, only four attribute computation rules were needed, two of which involved a call to `alloc()`. `alloc()` is a method which allocates space for a temporary result and returns a reference to it (in our examples, we will call these temporary results T1, T2, T3, etc). The attribute of an `ident` token is the value part of that token, indicating the run-time location for the variable.

Sample Problem 4.7.1

Show an attributed derivation tree for the expression `a+b` *using grammar G21.*

Solution:



4.7.1 Translating Expressions with Recursive Descent

In the recursive descent translator which follows, synthesized attributes of non-terminals are implemented as references, and inherited attributes are implemented as primitives. The alloc() and atom() methods are not shown here. alloc simply allocates space for a temporary result, and atom simply puts out

an atom, which could be a record consisting of four fields as described, above, in Section 4.7. Note that the underlying grammar of G21 is the grammar for expressions, G16, given in Section 4.4. The selection sets for this grammar are shown in that section. As in sample problem 4.6, we assume that the Token class has methods getClass() and getToken(). Also, we use our wrapper class, MutableInt, for synthesized attributes.

```
class Expressions
{
Token token;

static int next = 0;                    // for allocation of temporary
       // storage

public static void main (String[] args)
{
   Expresssions e = new Expressions();
      e.eval ();
}

void eval ()
{  MutableInt p = new MutableInt(0);
   token = new Token();

   token.getToken();

   Expr (p);                            // look for an expression
   if (token.getClass()!=Token.End) reject();
else accept();
}

void Expr (MutableInt p)
{  MutableInt q = new MutableInt(0);
   if (token.getClass()==Token.Lpar ||
       token.getClass()==Token.Ident
|| token.getClass()==Token.Num)

{  Term (q);                    // apply rule 1
   Elist (q.get(),p);
}                                  // end rule 1
   else reject();
}

void Elist (int p, MutableInt q)
{  int s;
```

```
   MutableInt r = new MutableInt();
   if (token.getClass()==Token.Plus)
{  token.getToken();             // apply rule 2
   Term (r);
   s = alloc();
   atom ("ADD", p, r, s);        // put out atom
   Elist (s,q);
}                                // end rule 2
   else if (token.getClass()==Token.End ||
token.getClass()==Token.Rpar)
q.set(p);                        // rule 3
   else reject();
}
void Term (MutableInt p)
{  MutableInt q = new MutableInt();
   if (token.getClass()==Token.Lpar
  || token.getClass()==Token.Ident
    || token.getClass()==Token.Num)
{  Factor (q);                   // apply rule 4
   Tlist (q.get(),p);
}                                // end rule 4
   else reject();
}

void Tlist (int p, MutableInt q)
{  int inp, s;
   MutableInt r = new MutableInt();
   if (token.getClass()==Token.Mult)
{  token.getToken();             // apply rule 5
   inp = token.getClass();
   Factor (r);
   s = alloc();
   atom ("MULT", p, r, s);
   Tlist (s,q);
}                               // end rule 5
   else if (token.getClass()==Token.Plus
  || token.getClass()==Token.Rpar
|| token.getClass()==Token.End)
q.set (p);             // rule 6
   else reject();
}

void Factor (MutableInt p)
{  if (token.getClass()==Token.Lpar)
{  token.getToken();           // apply rule 7
   Expr (p);
```

```
   if (token.getClass()==Token.Rpar)
token.getToken();
   else reject();
}                              // end rule 7
   else if (token.getClass()==Token.Ident
  || token.getClass()==Token.Num)
{  p.set(alloc());             // apply rule 8
   token.getToken();
}                              // end rule 8
   else reject();
}
```

4.7.2 Exercises

1. Show an attributed derivation tree for each of the following expressions, using grammar G21. Assume that the alloc method returns a new temporary location each time it is called (T1, T2, T3, ...).

   (a) a + b * c

   (b) (a + b) * c

   (c) (a)

   (d) a * b * c

2. In the recursive descent translator of Section 4.7.1, refer to the method `Tlist`. In it, there is the following statement:

   `Atom (MULT, p, r, s)`

   Explain how the three variables `p,r,s` obtain values before being put out by the atom method.

3. Improve grammar G21 to include the operations of subtraction and division, as well as unary plus and minus. Assume that there are SUB and DIV atoms to handle subtraction and division. Also assume that there is a NEG atom with two attributes to handle unary minus; the first is the expression being negated and the second is the result.

4.8 Decaf Expressions

Thus far we have seen how to translate simple infix expressions involving only addition and multiplication into a stream of atoms. Obviously, we also need to include subtraction and division operators in our language, but this is straightforward, since subtraction has the same precedence as addition and division has the same precedence as multiplication. In this section, we extend the notion of expression to include comparisons (boolean expressions) and assignments.

Boolean expressions are expressions such as $x > y$, or $y-2 == 33$. For those students familiar with C and C++, the comparison operators return ints (0=false, 1=true), but Java makes a distinction: the comparison operators return boolean results (true or false). If you have ever spent hours debugging a C or C++ program which contained `if (x=3)...` when you really intended `if (x==3) ...`, then you understand the reason for this change. The Java compiler will catch this error for you (assuming that x is not a boolean).

Assignment is also slightly different in Java. In C/C++ assignment is an operator which produces a result in addition to the side effect of assigning a value to a variable. A statement may be formed by following any expression with a semicolon. This is not the case in Java. The expression statement must be an assignment or a method call. Since there are no methods in Decaf, we're left with an assignment.

4.8.1 LBL, JMP, TST, and MOV atoms

At this point we need to introduce three new atoms indicated in Figure 4.17: LBL (label), JMP (jump, or unconditional branch), and TST (test, or conditional branch). A LBL atom is merely a placemarker in the stream of atoms that is put out. It represents a target location for the destination of a branch, such as JMP or TST. Ultimately, a LBL atom will represent the run-time memory address of an instruction in the object program. A LBL atom will always have one attribute which is a unique name for the label. A JMP atom is an unconditional branch; it must always be accompanied by a label name - the destination for the branch. A JMP atom will have one attribute - the name of the label which is the destination. A TST atom is used for a conditional branch. It will have four attributes: two attributes will identify the locations of two expressions being compared; another attribute will be a comparison code (1 is ==, 2 is <, ...); the fourth attribute will be the name of the label which is the destination for the branch. The intent is for the branch to occur at run time only if the result of comparing the two expressions with the given comparison operator is true.

To implement the assignment operator we will need a *MOV* atom which copies the value of one attribute to another:

$MOV_{source,,target}$ will move the value at at the source location to the target location at run time. (The omitted subscript is for compatibility with other atoms)

4.8.2 Boolean expressions

We will now explain the translation of boolean expressions. It turns out that every time we need to use a boolean expression in a statement, we really want to put out a TST atom that branches when the comparison is false.

In each case the source input is at the left, and the atoms to be put out are indented to the right. Rather than showing all the attributes at this point, their values are indicated with comments:

Atom	Attributes	Purpose
LBL	label name	Mark a spot to be used as a branch destination
JMP	label name	Unconditional branch to the label specified
TST	$Expr_1$	Compare $Expr_1$ and $Expr_2$ using the comparison code.
	$Expr_2$	Branch to the label if the result is true.
	comparison code	
	label name	

Figure 4.17: Atoms used for transfer of control

```
if
(x==3)
       [TST]                    // Branch to the Label only if
Stmt                            //   x==3 is false
       [Label]

////////////////////////////////////////////////////

while
       [Label1]
(x>2)
       [TST]                    // Branch to Label2 only if
 Stmt                           //   x>2 is false
       [JMP]                    // Unconditional branch to Label1
       [Label2]
```

Recall our six comparison codes; to get the logical complement of any comparison, we simply subtract the code from 7 as shown below:

```
Comparison     Code    Logical Complement    Code for complement
==             1       !=                    6
<              2       >=                    5
>              3       <=                    4
<=             4       >                     3
>=             5       <                     2
!=             6       ==                    1
```

Thus, to process a boolean expression all we need to do is put out a TST atom which allocates a new label name and branches to that label when the comparison is false (the label atom itself will be handled later, in section 4.9). Thus the attributed grammar rule for a boolean expression will be:

$BoolExpr_{Lbl} \rightarrow Expr_p\ compare_c\ Expr_q\ \{TST\}_{p,q,,7-c,Lbl}$

The TST atom represents a conditional branch in the object program. $\{TST\}_{a,b,,c,x}$ will compare the values stored at a and b, using the comparison whose code is c, and branch to a label designated x if the comparision is true. In the grammar rule above the attribute of BoolExpr, Lbl, is synthesized and represents the target label for the TST atom to branch in the object program. The attribute of the token compare, c, is an integer from 1-6 representing the comparison code. The use of comparison code 7-c inverts the sense of the comparison as desired.

4.8.3 Assignment

Next we handle assignment; an assignment is an operator which returns a result that can be used as part of a larger expression. For example:

```
x = (y = 2) + (z = 3);            // y is 2, z is 3, x is 5
```

Thus, an assignment operator does two things:

1. Assign the value of the right operand to the variable which is the left operand. This is a *side effect*, though a very important one, and it is the most common reason for using an assignment operator.
2. Return an explicit result: the value which was assigned. This is rarely used. The returned value is usually discarded.

This means that we will need to put out a MOV atom to implement the assignment, in addition to giving it a synthesized attribute to be moved up the tree. The left operand of the assignment must be an identifier, or what is often called an lvalue. Also, note that unlike the arithmetic operators, this operator associates to the right, which permits multiple assignments such as:

```
x = y = z = 0;                    // x, y, and, z are now all 0.
```

We could use a translation grammar such as the following:
$Expr_p \rightarrow AssignExpr_p$

$AssignExpr_p \rightarrow ident_p = Expr_q\{MOV\}_{q,,p}$
in which the location of the result is the same as the location of the identifier receiving the value. The attribute of the `Expr`, again, is synthesized. The output for an expression such as a = b + (c = 3) will be:

```
(MOV, 3,,c)
(ADD, b,c,T1)
(MOV, T1,,a)
```

An attributed translation grammar for Decaf expressions involving addition, subtraction, multiplication, division, comparisons, and assignment is shown below:

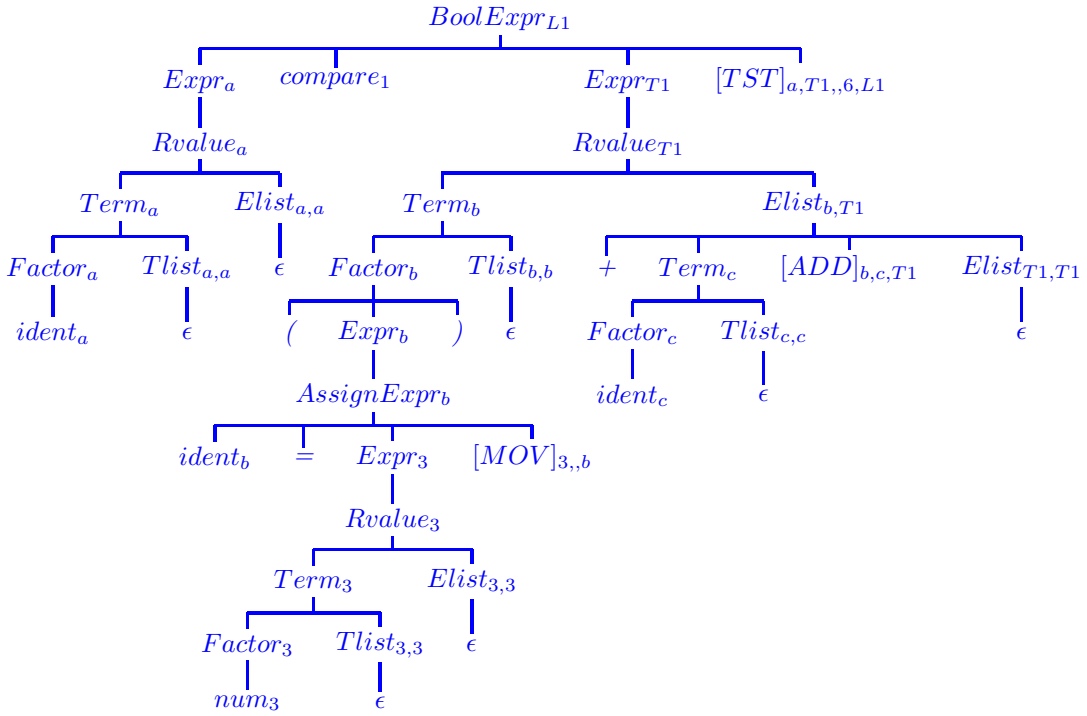
1.	$BoolExpr_{L1}$	$\rightarrow Expr_p compare_c Expr_q \{TST\}_{p,q,,7-c,L1}$	$L1 \leftarrow newLabel()$
2.	$Expr_p$	$\rightarrow AssignExpr_p$	
3.	$Expr_p$	$\rightarrow Rvalue_p$	
4.	$AssignExpr_p$	$\rightarrow ident_p = Expr_q \{MOV\}_{q,,p}$	
5.	$Rvalue_p$	$\rightarrow Term_q Elist_{q,p}$	
6.	$Elist_{p,q}$	$\rightarrow +Term_r \{ADD\}_{p,r,s} Elist_{s,q}$	$s \leftarrow alloc()$
7.	$Elist_{p,q}$	$\rightarrow -Term_r \{SUB\}_{p,r,s} Elist_{s,q}$	$s \leftarrow alloc()$
8.	$Elist_{p,q}$	$\rightarrow \epsilon$	$q \leftarrow p$
9.	$Term_p$	$\rightarrow Factor_q Tlist_{q,p}$	
10.	$Tlist_{p,q}$	$\rightarrow *Factor_r \{MUL\}_{p,r,s} Tlist_{s,q}$	$s \leftarrow alloc()$
11.	$Tlist_{p,q}$	$\rightarrow /Factor_r \{DIV\}_{p,r,s} Tlist_{s,q}$	$s \leftarrow alloc()$
12.	$Tlist_{p,q}$	$\rightarrow \epsilon$	$q \leftarrow p$
13.	$Factor_p$	$\rightarrow (Expr_p)$	
14.	$Factor_p$	$\rightarrow +Factor_p$	
15.	$Factor_p$	$\rightarrow -Factor_q \{Neg\}_{q,,p}$	$p \leftarrow alloc()$
16.	$Factor_p$	$\rightarrow num_p$	
17.	$Factor_p$	$\rightarrow ident_p$	

Note that the selection set for rule 2 is $\{ident\}$ and the selection set for rule 3 is $\{ident, (, num\}$. Since rules 2 and 3 define the same nonTerminal, this grammar is not LL(1). We can work around this problem by noticing that an assignment expression must have an assignment operator after the identifier. Thus, if we *peek ahead* one input token, we can determine whether to apply rule 2 or 3. If the next input token is an assignment operator, apply rule 2; if not, apply rule 3. This can be done with a slight modification to our token class - a `peek()` method which returns the next input token, but which has no effect on the next call to `getInput()`. The grammar shown above is said to be LL(2) because we can parse it top down by looking at no more than two input symbols at a time.

Sample Problem 4.8.1

Using the grammar for Decaf expressions given in this section, show an attributed derivation tree for the boolean expression `a == (b=3) + c .`

Solution:



4.8.4 Exercises

1. Show an attributed derivation tree using the grammar for Decaf expressions given in this section for each of the following expressions or boolean expressions (in part (a) start with Expr; in parts (b,c,d,e) start with BoolExpr):

   (a) a = b = c

   (b) a == b + c

   (c) (a=3) <= (b=2)

   (d) a == - (c = 3)

   (e) a * (b=3) + c != 9

2. Show the recursive descent parser for the nonterminals BoolExpr, Rvalue, and Elist given in the grammar for Decaf expressions. Hint: the selection sets for the first eight grammar rules are:

```
Sel (1)  =  {ident, num, (, +, -}
Sel (2)  =  {ident}
Sel (3)  =  {ident, num, (, +, -}
Sel (4)  =  {ident}
Sel (5)  =  {ident, num, (, +, -}
Sel (6)  =  {+}
Sel (7)  =  {-}
Sel (8)  =  {), N}
```

4.9 Translating Control Structures

In order to translate control structures, such as `for, while,` and `if` statements, we must first consider the set of primitive control operations available on the target machine. These are typically simple operations such as Unconditional Jump or Goto, Compare, and Conditional Jump. In order to implement these Jump operations, we need to establish a jump, or destination, address.

During the parse or syntax analysis phase there are, as yet, no machine addresses attached to the output. In addition, we must handle forward jumps when we do not know the destination of the jump. To solve this problem we introduce a special atom called a Label, which is used to mark the destination of a jump. During code generation, a machine address is associated with each Label atom. At this point, we need to add two additional fields to our atoms: one for comparison codes (1-6) and one for jump destinations.

We will use the following atoms to implement control structures:

```
JMP -  -  - -   Lbl     Unconditional jump to the specified label
TST E1 E2 - Cmp Lbl     Conditional branch if comparison is true
LBL -  -  - -   Lbl     Label used as branch destination
```

The purpose of the TST atom is to compare the values of expressions E1 and E2 using the specified comparison operator, Cmp, and then branch to the label Lbl if the comparison is true. The comparison operator will be the value part of the comparison token (there are six, shown below). For example, `TST,A,C,,4,L3` means jump to label L3 if A is less than or equal to C. The six comparison operators and codes are:

```
==     is 1          <=     is 4
<      is 2          >=     is 5
>      is 3          !=     is 6
```

The LBL atom is used as a tag or marker so that JMP and TST atoms can refer to a branch destination without knowing target machine addresses for these destinations.

In addition, there is one more atom which is needed for assignment statements; it is a Move atom, which simply assigns its source operand to its target operand:

```
MOV  Source - Target - -                    Target = Source
```

Figure 4.18 shows the sequence in which atoms are put out for the control structures for `while` and `for` statements (from top to bottom). Figure 4.19 shows the same information for the `if` statement. These figures show the input tokens, as well, so that you understand when the atoms are put out. The arrows indicate flow of control at run time; the arrows do not indicate the sequence in which the compiler puts out atoms. In Figures 4.18 and 4.19, the atoms are all enclosed in a boundary: ADD and JMP atoms are enclosed in rectangles, LBL atoms are enclosed in ovals, and TST atoms are enclosed in diamond shaped parallelograms.

The control structures in Figure 4.18 correspond to the following statement definitions:
1. $Stmt \rightarrow while(BoolExpr)Stmt$
2. $Stmt \rightarrow for(Expr; BoolExpr; Expr)Stmt$

The control structures in Figure 4.19 correspond to the following statement definitions:
3. $Stmt \rightarrow if(BoolExpr)Stmt\ ElsePart$
4. $ElsePart \rightarrow else\ Stmt$
5. $ElsePart \rightarrow \epsilon$

For the most part, Figures 4.18 and 4.19 are self explanatory. In Figure 4.19 we also show that a boolean expression always puts out a `TST` atom which branches if the comparison is false. The boolean expression has an attribute which is the target label for the branch. Note that for `if` statements , we must jump around the statement which is not to be executed. This provides for a relatively simple translation.

Unfortunately, the grammar shown above is not LL(1). This can be seen by finding the follow set of ElsePart, and noting that it contains the keyword `else`. Consequently, rules 4 and 5 do not have disjoint selection sets. However, it is still possible to write a recursive descent parser. This is due to the fact that all `else`s are matched with the closest preceeding unmatched `if`. When our parser for the nonterminal `ElsePart` encounters an `else`, it is never wrong to apply rule 4 because the closest preceeding unmatched `if` must be the one on top of the recursive call stack. Aho et. al.[1] claims that there is no LL(1) grammar for this language. This is apparently one of those rare instances where theory fails, but our practical knowledge of how things work comes to the rescue.

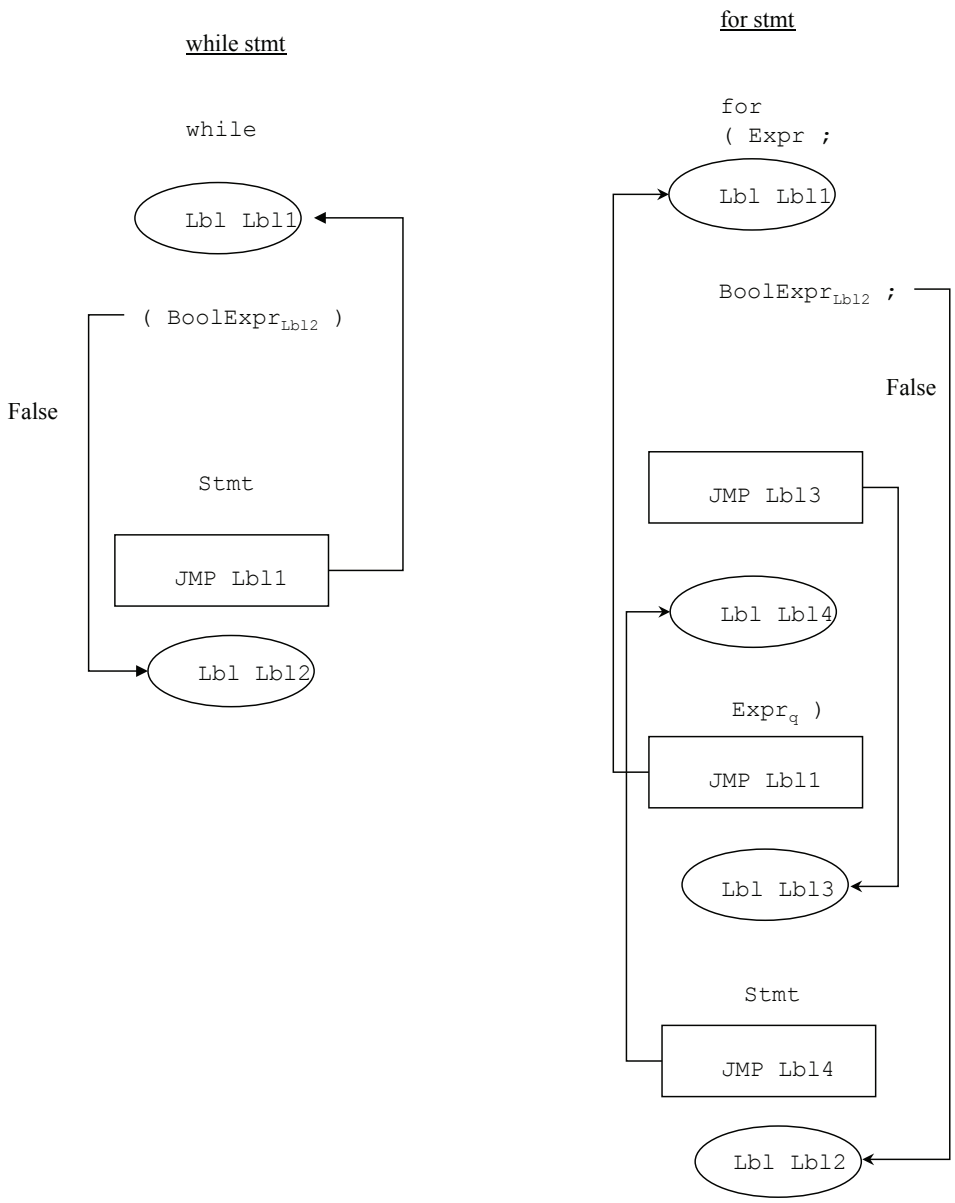


Figure 4.18: Control structures for `while` and `for` statements

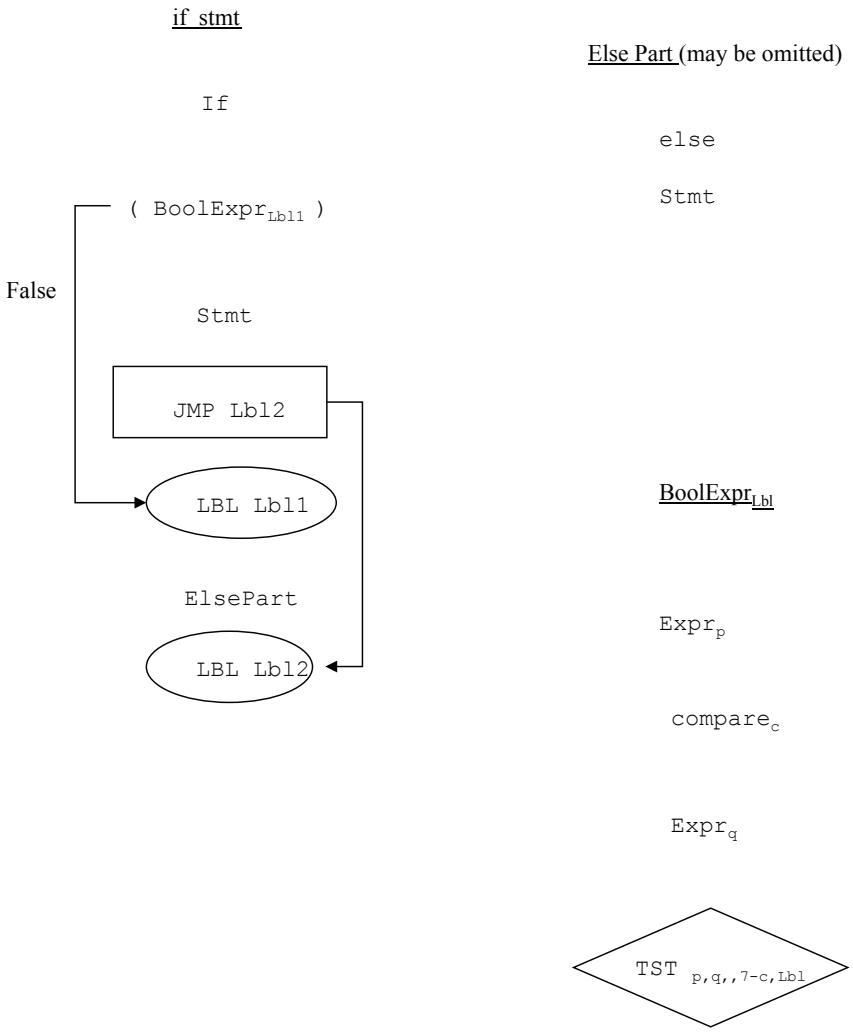


Figure 4.19: Control structures for `if` statements

The `for` statement described in Figure 4.18 requires some additional explanation. The following `for` statement and `while` statement are equivalent:

```
for (E1; boolean; E2)
        Stmt
```

```
E1 ;
while (boolean)
{ Stmt
 E2;
}
```

This means that after the atoms for the stmt are put out, we must put out a jump to the atoms corresponding to expression E2. In addition, there must be a jump after the atoms of expression E2 back to the beginning of the loop, boolean. The LL(2) grammar for Decaf shown in the next section makes direct use of Figures 4.18 and 4.19 for the control structures.

Sample Problem 4.9.1

Show the atom string which would be put out that corresponds to the following Java statement:
`while (x > 0) Stmt`

Solution:

```
(LBL, L1)
(TST,x,0,,4,L2)              // Branch to L2 if x<=0
```

Atoms for `Stmt`

```
(JMP,L1)
(LBL,L2)
```

4.9.1 Exercises

1. Show the sequence of atoms which would be put out according to Figures 4.18 and 4.19 for each of the following input strings:

(a)
```
if (a==b)
   while (x<y)
     Stmt
```

(b)
```
for (i = 1; i<=100; i = i+1)
         for (j = 1; j<=i; j = j+1)
             Stmt
```

(c)
```
if (a==b)
   for (i=1; i<=20; i=i+1)
       Stmt1
else
    while (i>0)
       Stmt2
```

(d)
```
 if (a==b)
    if (b>0)
       Stmt1
    else
       while (i>0)
          Stmt2
```

2. Show an attributed translation grammar rule for each of the control structures given in Figures 4.18 and 4.19. Assume `if` statements always have an `else` part and that there is a method, newlab, which allocates a new statement label.

   1. $WhileStmt \rightarrow while(BoolExpr)Stmt$
   2. $ForStmt \rightarrow for(AssignExpr; BoolExpr; AssignExpr)Stmt$
   3. $IfStmt \rightarrow if(BoolExpr)StmtelseStmt$

3. Show a recursive descent translator for your solutions to Problem 2. Show methods for `WhileStmt, ForStmt`, and `IfStmt`.

4. Does your Java compiler permit a loop control variable to be altered inside the loop, as in the following example?

```
for (int i=0; i<100; i = i+1)
      {  System.out.println (i);
         i = 100;
      }
```

4.10 Case Study: A Top Down Parser for Decaf

In this section we show how the concepts of this chapter can be applied to a compiler for a subset of Java, i.e. the Decaf language. We will develop a top down parser for Decaf and implement it using the technique known as recursive descent. The program is written in Java and is available at
`http://www.rowan.edu/~bergmann/books`. Note that in the software package there are actually two parsers: one is top down (for this chapter) and the other is bottom up (for Chapter 5). The bottom up parser is the one that is designed to work with the other phases to form a complete compiler. The top down parser is included only so that we can have a case study for this chapter. The SableCC grammar file, decaf.grammar, is used solely for lexical analysis at this point.

In order to write the recursive descent parser, we will need to work with a grammar that is LL. This grammar is relatively easy to write and is shown in Figure 4.20. The most difficult part of this grammar is the part which defines arithmetic expressions. It is taken directly from section 4.8.

The rest of the grammar is relatively easy because Java, like C and C++, was designed to be amenable to top-down parsing (in the tradition of Pascal); the developer of C++ recommends recursive descent parsers (see Stroustrup 1994). To see this, note that most of the constructs begin with key words. For example, when expecting a statement, the parser need examine only one token to decide which kind of statement it is. The possibilities are simply for, if, while, {, or identifier. In other words, the part of the Decaf grammar defining statements is simple (in the technical sense). Note that the C language permits a statement to be formed by appending a semicolon to any expression; Java, however, requires a simple statement to be either an assignment or a method call.

In Figure 4.20 there are two method calls: alloc() and newlab(). The purpose of alloc() is to find space for a temporary result, and the purpose of newlab() is to generate new label numbers, which we designate L1, L2, L3, ...

The control structures in Figure 4.20 are taken from Figures 4.18 and 4.19 and are described in section 4.9. As mentioned in that section, the convoluted logic flow in the `for` statement results from the fact that the third expression needs to be evaluated after the loop body is executed, and before testing for the next iteration.

The recursive descent parser is taken directly from Figure 4.20. The only departure is in the description of iterative structures such as IdentList and ArgList. Context-free grammars are useful in describing recursive constructs, but are not very useful in describing these iterative constructs. For example, the definition of IdentList is:
$IdentList \rightarrow identifier$
$IdentList \rightarrow identifier, IdentList$

While this is a perfectly valid definition of IdentList, it leads to a less efficient parser (for those compilers which do not translate tail recursion into loops).

$$
\begin{array}{lll}
Program & \rightarrow & classidentifier\{publicstaticvoidmain(String[]identifier)CompoundStmt\} \\
Declaration & \rightarrow & TypeIdentlist; \\
Type & \rightarrow & int \\
 & & |float \\
IdentList & \rightarrow & identifier, IdentList \\
 & & |identifier \\
Stmt & \rightarrow & AssignStmt \\
 & & |ForStmt \\
 & & |WhileStmt \\
 & & |IfStmt \\
 & & |CompoundStmt \\
 & & |Declaration \\
 & & |; \\
AssignStmt & \rightarrow & AssignExpr_p; \\
ForStmt & \rightarrow & for(OptAssignExpr_r; \{LBL\}_{Lbl1} OptBoolExpr_{Lbl4}; \{JMP\}_{Lbl2} \{LBL\}_{Lbl3} \\
 & & OptAssignExpr_r)\{JMP\}_{Lbl1} \{LBL\}_{Lbl2} Stmt \{JMP\}_{Lbl3} \{LBL\}_{Lbl4} \\
 & & Lbl1 \leftarrow newlab()\ Lbl2 \leftarrow newlab() \\
 & & Lbl3 \leftarrow newlab() \\
WhileStmt & \rightarrow & while\{LBL\}_{Lbl1}(BoolExpr_{Lbl2}) \\
 & & Stmt\{JMP\}_{Lbl1} \{LBL\}_{Lbl2} \\
 & & Lbl1 \leftarrow newlab() \\
IfStmt & \rightarrow & if(BoolExpr_{Lbl1}) \\
 & & Stmt\{JMP\}_{Lbl2} \{LBL\}_{Lbl1} ElsePart \{LBL\}_{Lbl2} \\
 & & Lbl2 \leftarrow newlab() \\
Elsepart & \rightarrow & elseStmt \\
 & & |\epsilon \\
CompoundStmt & \rightarrow & \{StmtList\} \\
 & & |\epsilon \\
StmtList & \rightarrow & StmtListStmt \\
 & & |\epsilon \\
OptAssignExpr & \rightarrow & AssignExpr_p \\
 & & |\epsilon \\
OptBoolExpr_{Lbl1} & \rightarrow & BoolExpr_{Lbl1} \\
 & & |\epsilon \\
BoolExpr_{Lbl1} & \rightarrow & BoolExpr_{Lbl1} \\
 & & Lbl1 \leftarrow newlab() \\
Expr_p & \rightarrow & AssignExpr_p \\
 & & |Rvalue_q Elist_{q,p} \\
AssignExpr_p & \rightarrow & identifier_p = Expr_q \{MOV\}_{q,,p} \\
Rvalue_p & \rightarrow & Term_q Elist_{p,q} \\
Elist_{p,q} & \rightarrow & +Term_r \{ADD\}_{p,r,s} Elist_{s,q} s \leftarrow alloc() \\
 & & |-Term_r \{SUB\}_{p,r,s} Elist_{s,q} s \leftarrow alloc() \\
 & & |\epsilon\ q \leftarrow p \\
Rvalue_p & \rightarrow & Factor_q Tlist_{p,q} \\
Tlist_{p,q} & \rightarrow & *Factor_r \{MUL\}_{p,r,s} Tlist_{s,q} s \leftarrow alloc() \\
 & & |/Factor_r \{DIV\}_{p,r,s} Tlist_{s,q} s \leftarrow alloc() \\
 & & |\epsilon\ q \leftarrow p \\
Factor_p & \rightarrow & (Expr_p) \\
Factor_p & \rightarrow & +Factor_p \\
Factor_p & \rightarrow & -Factor_q \{NEG\}_{q,,p} p \leftarrow alloc() \\
Factor_p & \rightarrow & num_p \\
Factor_p & \rightarrow & identifier_p
\end{array}
$$

Figure 4.20: An attributed translation grammar for Decaf

What we really want to do is use a loop to scan for a list of identifiers separated by commas. This can be done as follows:

```
if (token.getClass() != IDENTIFIER) error();
while (token.getClass() == IDENTIFIER)
      {  token.getToken();
         if (token.getClass() == ',')
            token.getToken();
      }
```

We use this methodology also for the methods ArgList() and StmtList().

Note that the fact that we have assigned the same attribute to certain symbols in the grammar, saves some effort in the parser. For example, the definition of Factor uses a subscript of p on the Factor as well as on the Expr, identifier, and number on the right side of the arrow. This simply means that the value of the Factor is the same as the item on the right side, and the parser is simply (ignoring unary operations):

```
void Factor (MutableInt p);
{      if (token.getClass() == '(' )
        {      token.getToken();
            Expr (p);
            if (token.getClass() == ')') token.getToken();
            else error();
        }
      else if (inp == IDENTIFIER)
        {        // store the value part of the identifier
            p.set (token.getValue());
            token.getToken();
        }
      else if (inp == NUMBER)
        {      p.set (token.getValue());
      token.getToken();
        }
      else  //    check for unary operators +, - ...
```

4.10.1 Exercises

1. Show the atoms put out as a result of the following Decaf statement:

```
if (a==3)
   { a = 4;
     for (i = 2; i<5; i=0 )
        i = i + 1;
   }
```

```
else
   while (a>5)
      i = i * 8;
```

2. Explain the purpose of each atom put out in our Decaf attributed translation grammar for the for statement:

   $ForStmt \rightarrow for(OptExpr_p; \{LBL\}_{Lbl1} OptBoolExpr_{Lbl3};$
   $\{JMP\}_{Lbl2} \{LBL\}_{Lbl4} OptExpr_r) \{JMP\}_{Lbl1}$
   $\{LBL\}_{Lbl2} Stmt \{JMP\}_{Lbl4} \{LBL\}_{Lbl3}$
   $Lbl1 \leftarrow newlab() Lbl2 \leftarrow newlab()$
   $Lbl3 \leftarrow newlab() Lbl4 \leftarrow newlab()$

3. The Java language has a *switch* statement.

   (a) Include a definition of the `switch` statement in the attributed translation grammar for Decaf.

   (b) Check your grammar by building an attributed derivation tree for a sample `switch` statement of your own design.

   (c) Include code for the `switch` statement in the recursive descent parser, decaf.java and parse.java .

4. Using the grammar of Figure 4.20, show an attributed derivation tree for the statement given in problem 1, above.

5. Implement a `do-while` statement in decaf, following the guidelines in problem 3.

4.11 Chapter Summary

A parsing algorithm reads an input string one symbol at a time, and determines whether the string is a member of the language of a particular grammar. In doing so, the algorithm uses a stack as it applies rules of the grammar. A top down parsing algorithm will apply the grammar rules in a sequence which corresponds to a downward direction in the derivation tree.

Not all context-free grammars are suitable for top down parsing. In general, the algorithm needs to be able to decide which grammar rule to apply without looking ahead at additional input symbols. We present an algorithm for finding selection sets, which are sets of input symbols, one set for each rule, and are used to direct the parser. Since the process of finding selection sets is fairly complex, we first define simple and quasi-simple grammars in which the process is easier.

We present two techniques for top down parsing: (1) pushdown machines and (2) recursive descent. These techniques can be used whenever all rules defining the same nonterminal have disjoint selection sets. We then define translation

grammars, which are capable of specifying output, and attributed grammars, in which it is possible for information to be passed from one grammar rule to another during the parse.

After learning how to apply these techniques to context-free grammars, we turn our attention to grammars for programming languages. In particular, we devise an attributed translation grammar for arithmetic expressions which can be parsed top down. In addition, we look at an attributed translation grammar for some of the common control structures: while, for, and if-else.

Finally, we examine a top down parser for our case study language: Decaf. This parser is written as a recursive descent parser in Java, and makes use of SableCC for the lexical scanner. In the interest of keeping costs down, it is not shown in the appendix; however, it is available along with the other software files at `http://www.rowan.edu/~bergmann/books`.

Chapter 5

Bottom Up Parsing

The implementation of parsing algorithms for LL(1) grammars , as shown in Chapter 4, is relatively straightforward. However, there are many situations in which it is not easy, if possible, to use an LL(1) grammar. In these cases, the designer may have to use a bottom up algorithm.

Parsing algorithms which proceed from the bottom of the derivation tree and apply grammar rules (in reverse) are called bottom up parsing algorithms. These algorithms will begin with an empy stack. One or more input symbols are moved onto the stack, which are then replaced by nonterminals according to the grammar rules. When all the input symbols have been read, the algorithm terminates with the starting nonterminal alone on the stack, if the input string is acceptable. The student may think of a bottom up parse as being similar to a derivation in reverse. Each time a grammar rule is applied to a sentential form, the rewriting rule is applied backwards. Consequently, derivation trees are constructed, or traversed, from bottom to top.

5.1 Shift Reduce Parsing

Bottom up parsing involves two fundamental operations. The process of moving an input symbol to the stack is called a shift operation, and the process of replacing symbols on the top of the stack with a nonterminal is called a reduce operation (it is a derivation step in reverse). Most bottom up parsers are called shift reduce parsers because they use these two operations. The following grammar will be used to show how a shift reduce parser works:

G22:

1. S $\rightarrow$ S a B
2. S $\rightarrow$ c
3. B $\rightarrow$ a b

A derivation tree for the string caabaab is shown in Figure 5.1. The shift reduce parser will proceed as follows: each step will be either a shift (shift an input

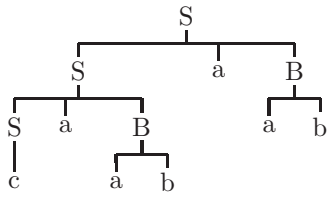


Figure 5.1: Derivation tree for the string `caabaab` using grammar G22

symbol to the stack) or reduce (reduce symbols on the stack to a nonterminal), in which case we indicate which rule of the grammar is being applied. The sequence of stack frames and input is shown in Figure 5.2, in which the stack frames are pictured horizontally to show, more clearly, the shifting of input characters onto the stack and the sentential forms corresponding to this parse. The algorithm accepts the input if the stack can be reduced to the starting nonterminal when all of the input string has been read.

Note in Figure 5.2 that whenever a reduce operation is performed, the symbols being reduced are always on top of the stack. The string of symbols being reduced is called a handle , and it is imperative in bottom up parsing that the algorithm be able to find a handle whenever possible. The bottom up parse shown in Figure 5.2 corresponds to the derivation shown below:

$S \Rightarrow \underline{SaB} \Rightarrow Sa\underline{ab} \Rightarrow \underline{SaB}aab \Rightarrow Sa\underline{ab}aab \Rightarrow \underline{c}aabaab$

Note that this is a right-most derivation; shift reduce parsing will always correspond to a right-most derivation. In this derivation we have underlined the handle in each sentential form. Read this derivation from right to left and compare it with Figure 5.2.

If the parser for a particular grammar can be implemented with a shift reduce algorithm, we say the grammar is LR (the L indicates we are reading input from the left, and the R indicates we are finding a right-most derivation). The shift reduce parsing algorithm always performs a reduce operation when the top of the stack corresponds to the right side of a rule. However, if the grammar is not LR, there may be instances where this is not the correct operation, or there may be instances where it is not clear which reduce operation should be performed. For example, consider grammar G23:

G23:

1. $S \rightarrow SaB$
2. $S \rightarrow a$
3. $B \rightarrow ab$

When parsing the input string aaab, we reach a point where it appears that we have a handle on top of the stack (the terminal a), but reducing that handle, as shown in Figure 5.3, does not lead to a correct bottom up parse. This is called a shift/reduce conflict because the parser does not know whether to shift an input symbol or reduce the handle on the stack. This means that the grammar is not LR, and we must either rewrite the grammar or use a different parsing

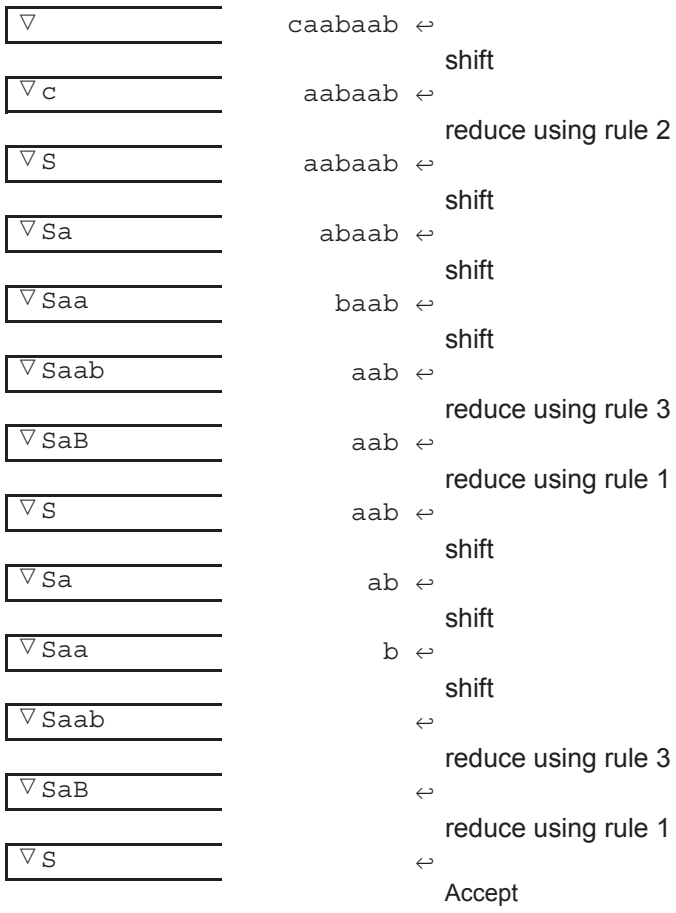


Figure 5.2: Sequence of stack frames parsing `caabaab` using grammar G22

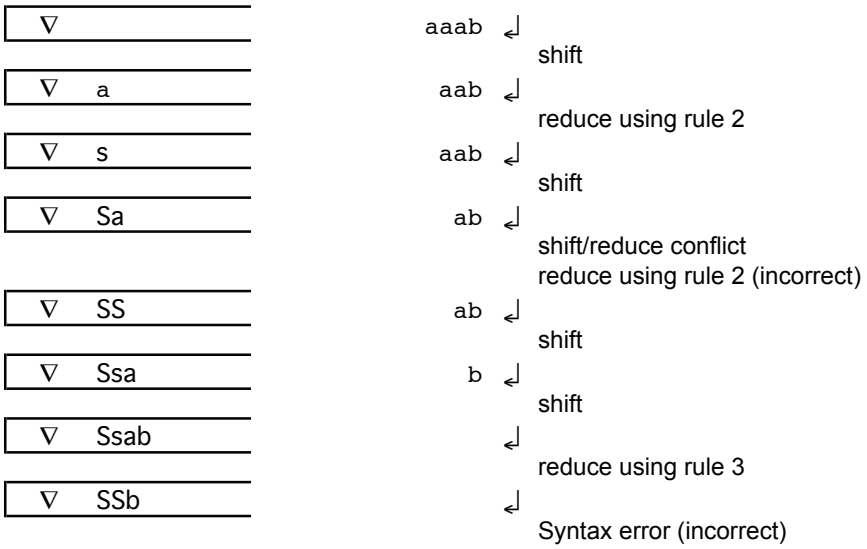


Figure 5.3: An example of a shift/reduce conflict leading to an incorrect parse using grammar G23

algorithm.

Another problem in shift reduce parsing occurs when it is clear that a reduce operation should be performed, but there is more than one grammar rule whose right hand side matches the top of the stack, and it is not clear which rule should be used. This is called a reduce/reduce conflict. Grammar G24 is an example of a grammar with a reduce/reduce conflict.

G24:

1. $S \rightarrow SA$
2. $S \rightarrow a$
3. $A \rightarrow a$

Figure 5.4 shows an attempt to parse the input string `aa` with the shift reduce algorithm, using grammar G24. Note that we encounter a reduce/reduce conflict when the handle `a` is on the stack because we don't know whether to reduce using rule 2 or rule 3. If we reduce using rule 2, we will get a correct parse, but if we reduce using rule 3 we will get an incorrect parse.

It is often possible to resolve these conflicts simply by making an assumption.

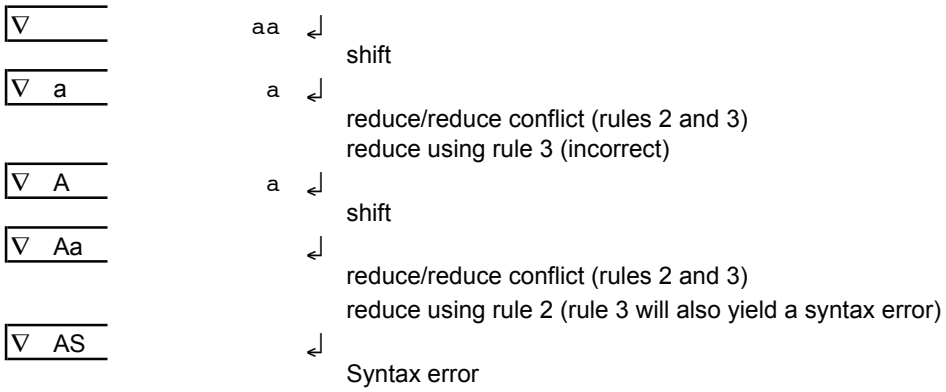


Figure 5.4: A reduce/reduce conflict using grammar G24

For example, all shift/reduce conflicts could be resolved by shifting rather than reducing. If this assumption always yields a correct parse, there is no need to rewrite the grammar.

In examples like the two just presented, it is possible that the conflict can be resolved by looking ahead at additional input characters. An LR algorithm that looks ahead k input symbols is called LR(k). When implementing programming languages bottom up, we generally try to define the language with an LR(1) grammar, in which case the algorithm will not need to look ahead beyond the current input symbol. An ambiguous grammar is not LR(k) for any value of k i.e. an ambiguous grammar will always produce conflicts when parsing bottom up with the shift reduce algorithm. For example, the following grammar for if statements is ambiguous:

1. Stmt → if (BoolExpr) Stmt else Stmt
2. Stmt → if (BoolExpr) Stmt

The BoolExpr in parentheses represents a true or false condition. Figure 5.5 shows two different derivation trees for the statement `if (BoolExpr) if (BoolExpr) Stmt else Stmt`. The tree on the right is the interpretation preferred by most programming languages (each else is matched with the closest preceding unmatched if). The parser will encounter a shift/reduce conflict when reading the else. The reason for the conflict is that the parser will be configured as shown in Figure 5.6.

In this case, the parser will not know whether to treat if (BoolExpr) Stmt as a handle and reduce it to Stmt according to rule 2, or to shift the else, which should be followed by a Stmt, thus reducing according to rule 1. However, if the parser can somehow be told to resolve this conflict in favor of the shift, then

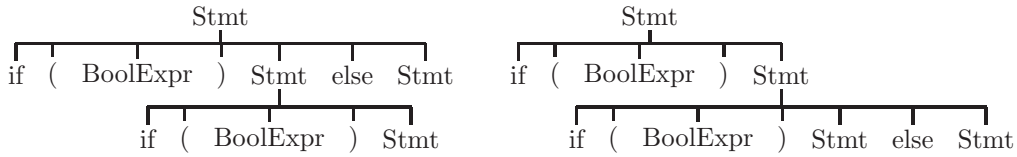


Figure 5.5: Two derivation trees for `if (BoolExpr) if (BoolExpr) Stmt else Stmt`

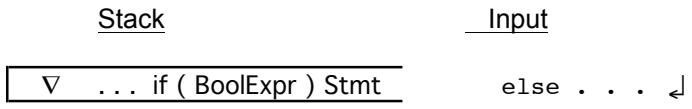


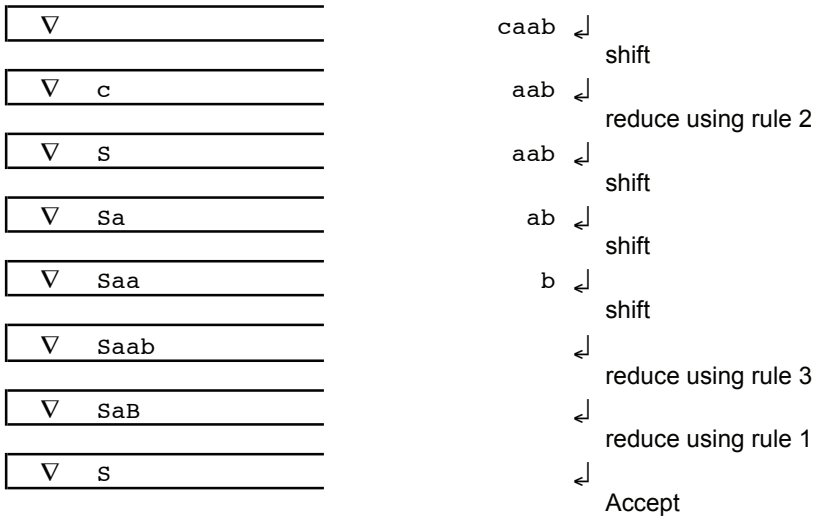
Figure 5.6: Parser configuration before reading the `else` part of an `if` statement

it will always find the correct interpretation. Alternatively, the ambiguity may be removed by rewriting the grammar, as shown in section 3.1.

Sample Problem 5.1.1

Show the sequence of stack and input configurations as the string `caab` *is parsed with a shift reduce parser, using grammar G22.*

Solution:



5.1.1 Exercises

1. For each of the following stack configurations, identify the handle using the grammar shown below:

   1. S → S A b
   2. S → a c b
   3. A → b B c
   4. A → b c
   5. B → b a
   6. B → A c

   (a) ▽ SSAb

   (b) ▽ SSbbc

   (c) ▽ SbBc

(d) 

2. Using the grammar of Problem 1, show the sequence of stack and input configurations as each of the following strings is parsed with shift reduce parsing:

   (a) acb
   (b) acbbcb
   (c) acbbbacb
   (d) acbbbcccb
   (e) acbbcbbcb

3. For each of the following input strings, indicate whether a shift/reduce parser will encounter a shift/reduce conflict, a reduce/reduce conflict, or no conflict when parsing, using the grammar below:

   1. S → S a b
   2. S → b A
   3. A → b b
   4. A → b A
   5. A → b b c
   6. A → c

   (a) b c
   (b) b b c a b
   (c) b a c b

4. Assume that a shift/reduce parser always chooses the lower numbered rule (i.e., the one listed first in the grammar) whenever a reduce/reduce conflict occurs during parsing, and it chooses a shift whenever a shift/reduce conflict occurs. Show a derivation tree corresponding to the parse for the sentential form `if (BoolExpr) if (BoolExpr) Stmt else Stmt`, using the following ambiguous grammar. Since the grammar is not complete, you may have nonterminal symbols at the leaves of the derivation tree.
   1. Stmt → if (BoolExpr) Stmt else Stmt
   2. Stmt → if (BoolExpr) Stmt

5.2 LR Parsing With Tables

One way to implement shift reduce parsing is with tables that determine whether to shift or reduce, and which grammar rule to reduce. This technique makes use of two tables to control the parser. The first table, called the action table,

determines whether a shift or reduce is to be invoked. If it specifies a reduce, it also indicates which grammar rule is to be reduced. The second table, called a goto table, indicates which stack symbol is to be pushed on the stack after a reduction. A shift action is implemented by a push operation followed by an advance input operation. A reduce action must always specify the grammar rule to be reduced. The reduce action is implemented by a Replace operation in which stack symbols on the right side of the specified grammar rule are replaced by a stack symbol from the goto table (the input pointer is retained). The symbol pushed is not necessarily the nonterminal being reduced, as shown below. In practice, there will be one or more stack symbols corresponding to each nonterminal.

The columns of the goto table are labeled by nonterminals, and the the rows are labeled by stack symbols. A cell of the goto table is selected by choosing the column of the nonterminal being reduced and the row of the stack symbol just beneath the handle.

For example, suppose we have the following stack and input configuration:

Stack	Input
▽ S	ab↩

in which the bottom of the stack is to the left. The action shift will result in the following configuration:

Stack	Input
▽ Sa	b↩

The a has been shifted from the input to the stack. Suppose, then, that in the grammar, rule 7 is:

7. B → Sa

Select the row of the goto table labeled ▽ and the column labeled B. If the entry in this cell is *push X*, then the action *reduce 7* would result in the following configuration:

Stack	Input
▽ X	b↩

Figure 5.7 shows the LR parsing tables for grammar G5 for arithmetic expressions involving only addition and multiplication (see section 3.1). As in previous pushdown machines, the stack symbols label the rows, and the input symbols label the columns of the action table. The columns of the goto table are labeled by the nonterminal being reduced. The stack is initialized with a ▽ symbol to mark the bottom of the statck, and blank cells in the action table indicate syntax errors in the input string. Figure 5.8 shows the sequence of configurations which would result when these tables are used to parse the input string (var+var)*var.

	Action Table					
	+	*	(	)	var	↩
▽			shift (		shift var	
Expr1	shift +					Accept
Term1	reduce 1	shift *		reduce 1		reduce 1
Factor3	reduce 3	reduce 3		reduce 3		reduce 3
(			shift (		shift var	
Expr5	shift +			shift)		
)	reduce 5	reduce 5		reduce 5		reduce 5
+			shift (		shift var	
Term2	reduce 2	shift *		reduce 2		reduce 2
*			shift (		shift var	
Factor4	reduce 4	reduce 4		reduce 4		reduce 4
var	reduce 6	reduce 6		reduce 6		reduce 6

	GoTo Table		
	Expr	Term	Factor
▽	push Expr1	push Term2	push Factor4
Expr1			
Term1			
Factor3			
(	push Expr5	push Term2	push Factor4
Expr5			
)			
+		push Term1	push Factor4
Term2			
*			push Factor3
Factor4			
var			

Initial Stack

Figure 5.7: Action and Goto tables to parse simple arithmetic expressions

Stack	Input	Action	Goto
▽	(var+var)*var ↩		
		shift (	
▽(	var+var)*var ↩		
		shift var	
▽(var	+var)*var ↩		
		reduce 6	push Factor4
▽(Factor4	+var)*var ↩		
		reduce 4	push Term2
▽(Term2	+var)*var ↩		
		reduce 2	push Expr5
▽(Expr5	+var)*var ↩		
		shift +	
▽(Expr5+	var)*var ↩		
		shift var	
▽(Expr5+var	)*var ↩		
		reduce 6	push Factor4
▽(Expr5+Factor4	)*var ↩		
		reduce 4	push Term1
▽(Expr5+Term1	)*var ↩		
		reduce 1	push Expr5
▽(Expr5	)*var ↩		
		shift)	
▽(Expr5)	*var ↩		
		reduce 5	push Factor4
▽Factor4	*var ↩		
		reduce 4	push Term2
▽Term2	*var ↩		
		shift *	
▽Term2*	var ↩		
		shift var	
▽Term2*var	↩		
		reduce 6	push Factor3
▽Term2*Factor3	↩		
		reduce 3	push Term2
▽Term2	↩		
		reduce 2	push Expr1
▽Expr1	↩		
		Accept	

Figure 5.8: Sequence of configurations when parsing `(var+var)*var`

G5:

1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor → (Expr)
6. Factor → var

The operation of the LR parser can be described as follows:

```
1.  Find the action corresponding to the current input and the top stack symbol.
2.  If that action is a shift action:
    a.  Push the input symbol onto the stack.
    b.  Advance the input pointer.
3.  If that action is a reduce action:
    a.  Find the grammar rule specified by the reduce action.
    b.  The symbols on the right side of the rule should also be on the top of the
        stack --  pop them all off the stack.
    c.  Use the nonterminal on the left side of the grammar rule to indicate a
        column of the goto table, and use the top stack symbol to indicate a row
        of the goto table.  Push the indicated stack symbol onto the stack.
    d.  Retain the input pointer.
4.  If that action is blank, a syntax error has been detected.
5.  If that action is Accept, terminate.
6.  Repeat from step 1.
```

Sample Problem 5.2.1

Show the sequence of stack, input, action, and goto configurations for the input `var*var` *using the parsing tables of Figure 5.7.*

Solution:

Stack	Input	Action	Goto
▽	var*var ↩		
		shift var	
▽var	*var ↩		
		reduce 6	push Factor4
▽Factor4	*var ↩		
		reduce 4	push Term2
▽Term2	*var ↩		
		shift *	
▽Term2*	var ↩		
		shift var	
▽Term2*var	↩		
		reduce 6	push Factor3
▽Term2*Factor3	↩		
		reduce 3	push Term2
▽Term2	↩		
		reduce 2	push Expr1
▽Expr1	↩		
		Accept	

There are three principle techniques for constructing the LR parsing tables. In order from simplest to most complex or general, they are called: Simple LR (SLR), Look Ahead LR (LALR), and Canonical LR (LR). SLR is the easiest technique to implement, but works for a small class of grammars. LALR is more difficult and works on a slightly larger class of grammars. LR is the most general, but still does not work for all unambiguous context free grammars. In all cases, they find a rightmost derivation when scanning from the left (hence LR). These techniques are beyond the scope of this text, but are described in Parsons [17] and Aho et. al. [1].

5.2.1 Exercises

1. Show the sequence of stack and input configurations and the reduce and goto operations for each of the following expressions, using the *action* and *goto* tables of Figure 5.7.

   (a) var

   (b) (var)

   (c) var + var * var

   (d) (var*var) + var

   (e) (var * var

5.3 SableCC

For many grammars, the LR parsing tables can be generated automatically from the grammar. There are several software systems designed to generate a parser automatically from specifications (as mentioned in section 2.4). In this chapter we will be using software developed at McGill University, called SableCC.

5.3.1 Overview of SableCC

SableCC is described well in the thesis of its creator, Etienne Gagnon [10] (see www.sablecc.org). The user of SableCC prepares a grammar file, as described in section 2.4, as well as two java classes: Translation and Compiler. These are stored in the same directory as the parser, lexer, node, and analysis directories. Using the grammar file as input, SableCC generates java code the purpose of which is to compile source code as specified in the grammar file. SableCC generates a lexer and a parser which will produce an abstract syntax tree as output. If the user wishes to implement actions with the parser, the actions are specified in the Translation class. An overview of this software system is presented in Figure 5.9.

5.3.2 Structure of the SableCC Source Files

The input to SableCC is called a grammar file. This file contains the specifications for lexical tokens, as well as syntactic structures (statements, expressions, ...) of the language for which we wish to construct a compiler. Neither actions nor attributes are included in the grammar file. There are six sections in the grammar file:

```
1.      Package
2.      Helpers
3.      States
4.      Tokens
5.      Ignored Tokens
6.      Productions
```

The first four sections were described in section 2.4. The `Ignored Tokens` section gives you an opportunity to specify tokens that should be ignored by the parser (typically white space and comments). The Productions section contains the grammar rules for the language being defined. This is where syntactic structures such as statements, expressions, etc. are defined. Each definition consists of the name of the syntactic type being defined (i.e. a nonterminal), an equal sign, an EBNF definition, and a semicolon to terminate the production. As mentioned in section 2.4, all names in this grammar file must be lower case. An example of a production defining a while statement is shown below (l_par and r_par are left parenthesis and right parenthesis tokens, respectively):

```
stmt = while l_par bool_expr r_par stmt ;
```

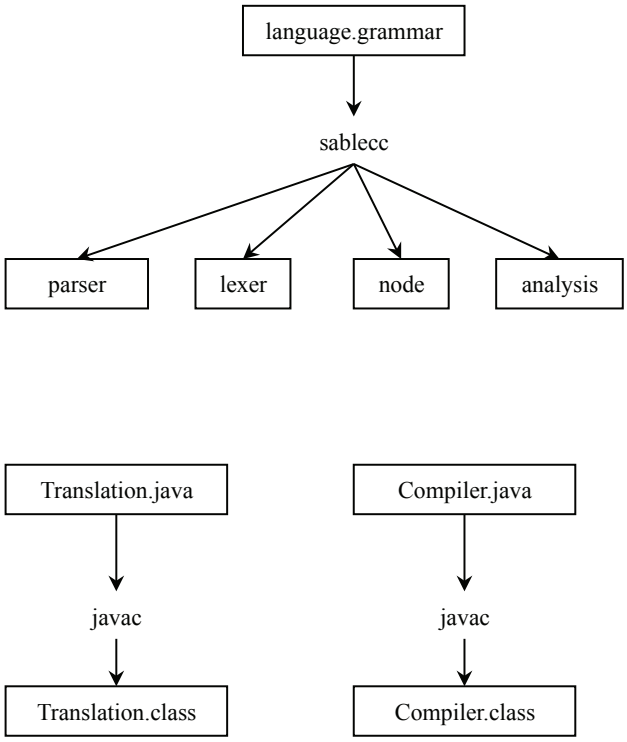


Figure 5.9: Generation and compilation of a compiler using SableCC

Note that the semicolon at the end is not the token for a semicolon, but a terminator for the stmt rule. Productions may use EBNF-like constructs. If x is any grammar symbol, then:

```
x?            // An optional x (0 or 1 occurrences of x)
x*            // 0 or more occurrences of x
x+            // 1 or more occurrences of x
```

Alternative definitions, using |, are also permitted. However, alternatives must be labeled with names enclosed in braces. The following defines an argument list as 1 or more identifiers, separated with commas:

```
arg_list =    {single}  identifier
            | {multiple} identifier ( comma identifier ) +
            ;
```

The names single and multiple enable the user to refer to one of these alternatives when applying actions in the Translation class. Labels must also be used when two identical names appear in a grammar rule. Each item label must be enclosed in brackets, and followed by a colon:

```
for_stmt =   for l_par [init]: assign_expr semi bool_expr
                 semi [incr]: assign_expr r_par stmt ;
```

Since there are two occurrences of assign_expr in the above definition of a `for` statement, they must be labeled. The first is labeled `init`, and the second is labeled `incr`.

5.3.3 An Example Using SableCC

The purpose of this example is to translate infix expressions involving addition, subtraction, multiplication, and division into postfix expressions, in which the operations are placed after both operands. Note that parentheses are never needed in postfix expressions, as shown in the following examples:

```
Infix                          Postfix
2 + 3 * 4                      2 3 4 * +
2 * 3 + 4                      2 3 * 4 +
( 2 + 3 ) * 4                  2 3 + 4 *
2 + 3 * ( 8 - 4 ) - 2          2 3 8 4 - * + 2 -
```

There are four sections in the grammar file for this program. The first section specifies that the package name is 'postfix'. All java software for this program will be part of this package. The second section defines the tokens to be used.

No Helpers are needed, since the numbers are simple whole numbers, specified as one or more digits. The third section specifies that blank (white space) tokens are to be ignored; this includes tab characters and newline characters. Thus the user may input infix expressions in free format. The fourth section, called Productions, defines the syntax of infix expressions. It is similar to the grammar given in section 3.1, but includes subtraction and division operations. Note that each alternative definition for a syntactic type must have a label in braces. The grammar file is shown below:

```
Package postfix;

Tokens
 number = ['0'..'9']+;
 plus =    '+';
 minus =   '-';
 mult =    '*';
 div =     '/';
 l\_par =    '(';
 r\_par =    ')';
 blank =   (' ' | 10 | 13 | 9)+ ;
 semi =    ';' ;

Ignored Tokens
 blank;

Productions
 expr =
        {term}   term |
        {plus}   expr plus term |
        {minus}  expr minus term
        ;
 term =
        {factor}  factor |
        {mult}    term mult factor |
        {div}     term div factor
        ;
 factor =
        {number}  number |
        {paren}   l_par expr r_par
        ;
```

Now we wish to include actions which will put out postfix expressions. SableCC will produce parser software which will create an abstract syntax tree for a particular infix expression, using the given grammar. SableCC will also

produce a class called DepthFirstAdapter, which has methods capable of visiting every node in the syntax tree. In order to implement actions, all we need to do is extend DepthFirstAdapter (the extended class is usually called Translation), and override methods corresponding to rules (or tokens) in our grammar. For example, since our grammar contains an alternative, Mult, in the definition of Term, the DepthFirstAdapter class contains a method named outAMultTerm. It will have one parameter which is the node in the syntax tree corresponding to the Term. Its signature is

```
public void outAMultTerm (AMultTerm node)
```

This method will be invoked when this node in the syntax tree, and all its descendants, have been visited in a depth-first traversal. In other words, a Term, consisting of a Term, a mult (i.e. a '*'), and a Factor have been successfully scanned. To include an action for this rule, all we need to do is override the outAMultTerm method in our extended class (Translation). In our case we want to print out a '+' after scanning a '+' and both of its operands. This is done by overriding the outAPlusExpr method. When do we print out a number? This is done when a number is seen in the {number} alternative of the definition of factor. Therefore, override the method outANumberFactor. In this method all we need to do is print the parameter node (all nodes have toString() methods, and therefore can be printed). The Translation class is shown below:

```
package postfix;
import postfix.analysis.*;  // needed for DepthFirstAdapter
import postfix.node.*;      // needed for syntax tree nodes.

class Translation extends DepthFirstAdapter
{
public void outAPlusExpr(APlusExpr node)
{// out of alternative {plus} in expr, we print the plus.
   System.out.print ( " + ");
}

public void outAMinusExpr(AMinusExpr node)
{// out of alternative {minus} in expr, we print the minus.
   System.out.print ( " - ");
}

public void outAMultTerm(AMultTerm node)
{// out of alternative {mult} in term, we print the minus.
   System.out.print ( " * ");
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in term, we print the minus.
   System.out.print ( " / ");
```

```
}

public void outANumberFactor (ANumberFactor node)
//  out of alternative {number} in factor, we print the number.
{   System.out.print (node + " "); }
}
```

There are other methods in the DepthFirstAdapter class which may also be overridden in the Translation class, but which were not needed for this example. They include the following:

- There is an 'in' method for each alternative, which is invoked when a node is about to be visited. In our example, this would include the method `public void inAMultTerm (AMultTerm node)`
- There is a 'case' method for each alternative. This is the method that visits all the descendants of a node, and it is not normally necessary to override this method. An example would be `public void caseAMultTerm (AMultTerm node)`
- There is also a 'case' method for each token; the token name is prefixed with a 'T' as shown below:

```
public void caseTNumber (TNumber token)
{       // action for number tokens       }
```

An important problem to be addressed is how to invoke an action in the middle of a rule (an embedded action). Consider the while statement definition:

```
while_stmt = {while} while l_par bool_expr r_par stmt ;
```

Suppose we wish to put out a LBL atom after the while keyword token is seen. There are two ways to do this. The first way is to rewrite the grammar, and include a new nonterminal for this purpose (here we call it `while_token`):

```
while_stmt =   {while} while_token l_par
                 bool_expr r_par stmt ;
while_token =  while ;
```

Now the method to be overridden could be:

```
public void outAWhileToken (AWhileToken node)
{  System.out.println ("LBL") ;  } // put out a LBL atom.
```

The other way to solve this problem would be to leave the grammar as is and override the case method for this alternative. The case methods have not been explained in full detail, but all the user needs to do is to copy the case method from DepthFirstAdapter, and add the action at the appropriate place. In this example it would be:

```
public void caseAWhileStmt (AWhileStmt node)
    {   inAWhileStmt(node);
        if(node.getWhile() != null)
          {  node.getWhile().apply(this)  }
      ////////////// insert action here /////////////////
      System.out.println ("LBL");      // embedded action
      ////////////////////////////////////////////////
        if(node.getLPar() != null)
          {  node.getLPar().apply(this);  }
        if(node.getBoolExpr() != null)
          {   node.getBoolExpr().apply(this);   }
        if(node.getRPar() != null)
          {   node.getRPar().apply(this);  }
        if (node.getStmt() != null)
          {  node.getStmt().apply (this) ;  }
        outAWhileStmt (node);
    }
```

The student may have noticed that SableCC tends to alter names that were included in the grammar. This is done to prevent ambiguities. For example, l_par becomes LPar, and bool_expr becomes BoolExpr.

In addition to a Translation class, we also need a Compiler class. This is the class which contains the main method, which invokes the parser. The Compiler class is shown below:

```
package postfix;
import postfix.parser.*;
import postfix.lexer.*;
import postfix.node.*;
import java.io.*;

public class Compiler
{
public static void main(String[] arguments)
{ try
  { System.out.println("Type one expression");

    // Create a Parser instance.
```

```
    Parser p = new Parser
        ( new Lexer
          ( new PushbackReader
            ( new InputStreamReader(System.in), 1024)));

    // Parse the input.
    Start tree = p.parse();

    // Apply the translation.
    tree.apply(new Translation());

    System.out.println();
  }
  catch(Exception e)
    { System.out.println(e.getMessage()); }
}
}
```

This completes our example on translating infix expressions to postfix. The source code is available at `http://www.rowan.edu/~bergmann/books`. In section 2.3 we discussed the use of hash tables in lexical analysis. Here again we make use of hash tables, this time using the Java class HashMap (from java.util). This is a general storage-lookup table for any kind of objects. Use the `put` method to store an object, with a key:

```
void put (Object key, Object value);
```

and use the `get` method to retrieve a value from the table:

```
Object get (Object key)
```

Sample Problem 5.3.1

Use SableCC to translate infix expressions involving addition, subtraction, multiplication, and division of whole numbers into atoms. Assume that each number is stored in a temporary memory location when it is encountered. For example, the following infix expression:

```
34 + 23 * 8 - 4
```

should produce the list of atoms:

```
MUL T2 T3 T4
ADD T1 T4 T5
SUB T5 T6 T7
```

Here it is assumed that 34 is stored in T1, 23 is stored in T2, 8 is stored in T3, and 4 is stored in T6.

Solution:

Since we are again dealing with infix expressions, the grammar given in this section may be reused. Simply change the package name to exprs.

The Compiler class may also be reused as is. All we need to do is rewrite the Translation class.

To solve this problem we will need to allocate memory locations for sub-expressions and remember where they are. For this purpose we use a java Map. A Map stores key-value pairs, where the key may be any object, and the value may be any object. Once a value has been stored (with a put method), it can be retrieved with its key (using the get method). In our Map, the key will always be a Node, and the value will always be an Integer. The Translation class is shown below:

```
package exprs;
import exprs.analysis.*;
import exprs.node.*;
import java.util.*; // for Hashtable
import java.io.*;

class Translation extends DepthFirstAdapter
{
// Use a Map to store the memory locations for exprs
// Any node may be a key, its memory location will be the
//     value, in a (key,value) pair.

Map <Node, Integer> hash = new HashMap <Node, Integer>();

public void caseTNumber(TNumber node)
// Allocate memory loc for this node, and put it into
//  the map.
{ hash.put (node, alloc()); }

public void outATermExpr (ATermExpr node)
{  // Attribute of the expr same as the term
    hash.put (node, hash.get(node.getTerm()));
}
```

```
public void outAPlusExpr(APlusExpr node)
{// out of alternative {plus} in Expr, we generate an
 //  ADD atom.
   int i = alloc();
   hash.put (node, i);
   atom ("ADD", (Integer)hash.get(node.getExpr()),
(Integer)hash.get(node.getTerm()), i);
}

public void outAMinusExpr(AMinusExpr node)
{// out of alternative {minus} in Expr,
 // generate a minus atom.
    int i = alloc();
    hash.put (node, i);
    atom ("SUB", (Integer)hash.get(node.getExpr()),
(Integer)hash.get(node.getTerm()), i);
}

public void outAFactorTerm (AFactorTerm node)
{ //  Attribute of the term same as the factor
        hash.put (node, hash.get(node.getFactor()));
 }

public void outAMultTerm(AMultTerm node)
{// out of alternative {mult} in Factor, generate a mult
 // atom.
    int i = alloc();
    hash.put (node, i);
    atom ("MUL", (Integer)hash.get(node.getTerm()),
                 (Integer) hash.get(node.getFactor()) , i);
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in Factor,
 // generate a div atom.
    int i = alloc();
    hash.put (node, i);
    atom ("DIV", (Integer) hash.get(node.getTerm()),
                 (Integer) hash.get(node.getFactor()), i);
}

public void outANumberFactor (ANumberFactor node)
{   hash.put (node, hash.get (node.getNumber())); }

public void outAParenFactor (AParenFactor node)
```

```
{   hash.put (node, hash.get (node.getExpr())); }

void atom (String atomClass, Integer left, Integer right,
           Integer result)
{   System.out.println (atomClass +  " T" + left + " T"  +
right +  " T" + result);
}

static int avail = 0;

int alloc()
{ return ++avail; }

}
```

5.3.4 Exercises

1. Which of the following input strings would cause this SableCC program to produce a syntax error message?

```
Tokens
 a = 'a';
 b = 'b';
 c = 'c';
 newline = [10 + 13];
Productions
  line = s newline ;
  s =           {a1} a s b
              | {a2} b w c
              ;
  w =           {a1} b w b
              | {a2} a c
              ;
```

   (a) bacc (b) ab (c) abbacbcb (d) bbacbc (e) bbacbb

2. Using the SableCC program from problem 1, show the output produced by each of the input strings given in Problem 1, using the Translation class shown below.

```
package ex5_3;
import ex5_3.analysis.*;
import ex5_3.node.*;
import java.util.*;
import java.io.*;

class Translation extends DepthFirstAdapter
{

public void outAA1S (AA1S node)
{  System.out.println ("rule 1"); }

public void outAA2S (AA2S node)
{  System.out.println ("rule 2"); }

public void outAA1W (AA1W node)
{  System.out.println ("rule 3"); }

public void outAA2W (AA2W node)
{  System.out.println ("rule 4"); }
}
```

3. A Sexpr is an atom or a pair of Sexprs enclosed in parentheses and separated with a period. For example, if A, B, C, ...Z and NIL are all atoms, then the following are examples of Sexprs:

   A (A.B) ((A.B).(B.C)) (A.(B.(C.NIL)))

   A List is a special kind of Sexpr. A List is the atom NIL or a List is a dotted pair of Sexprs in which the first part is an atom or a List and the second part is a List. The following are examples of lists:

   NIL (A.NIL) ((A.NIL).NIL) ((A.NIL).(B.NIL)) (A.(B.(C.NIL)))

   (a) Show a SableCC grammar that defines a Sexpr.

   (b) Show a SableCC grammar that defines a List.

   (c) Add a Translation class to your answer to part (b) so that it will print out the total number of atoms in a List. For example:

   ((A.NIL).(B.(C.NIL))) 5 atoms

4. Use SableCC to implement a syntax checker for a typical database command language. Your syntax checker should handle at least the following kinds of commands:

   ```
   RETRIEVE employee_file
   PRINT
   ```

```
DISPLAY FOR salary >= 1000000
PRINT FOR "SMITH" = lastname
```

5. The following SableCC grammar and Translation class are designed to implement a simple desk calculator with the standard four arithmetic functions (it uses floating-point arithmetic only). When compiled and run, the program will evaluate a list of arithmetic expressions, one per line, and print the results. For example:

```
2+3.2e-2
2+3*5/2
(2+3)*5/2
16/(2*3 - 6*1.0)
       2.032
       9.5
       12.5
       infinity
```

Unfortunately, the grammar and Java code shown below are incorrect. There are four mistakes, some of which are syntactic errors in the grammar; some of which are syntactic Java errors; some of which cause run-time errors; and some of which don't produce any error messages, but do produce incorrect output. Find and correct all four mistakes. If possible, use a computer to help debug these programs.

The grammar, exprs.grammar is shown below:

```
Package exprs;

Helpers
 digits = ['0'..'9']+ ;
 exp =    ['e' + 'E'] ['+' + '-']? digits ;
Tokens
 number = digits '.'? digits? exp? ;
 plus =    '+';
 minus =   '-';
 mult =    '*';
 div =     '/';
 l_par =    '(';
 r_par =    ')';
 newline = [10 + 13] ;
 blank =   (' ' | 't')+;
 semi =    ';' ;

Ignored Tokens
```

```
 blank;

Productions
 exprs =    expr newline
        |   exprs embed
        ;
 embed = expr newline;
 expr =
        {term}   term |
        {plus}   expr plus term |
        {minus}  expr minus term
        ;
 term =
        {factor}  factor |
        {mult}    term mult factor |
        {div}     term div factor |
        ;
 factor =
        {number}  number |
        {paren}   l_par expr r_par
        ;
```

The Translation class is shown below:

```
package exprs;
import exprs.analysis.*;
import exprs.node.*;
import java.util.*;

class Translation extends DepthFirstAdapter
{
Map <Node, Integer> hash =
new HashMap <Node, Integer> ();  // store expr values

public void outAE1Exprs (AE1Exprs node)
{  System.out.println ("   " + getVal (node.getExpr())); }

public void outAEmbed (AEmbed node)
{  System.out.println ("   " + getVal (node.getExpr())); }

public void caseTNumber(TNumber node)
{ hash.put (node, new Double (node.toString())) ; }

public void outAPlusExpr(APlusExpr node)
```

```
{// out of alternative {plus} in Expr, we add the
 // expr and the term
   hash.put (node, new Double (getPrim (node.getExpr())
               + getPrim(node.getTerm())));
}

public void outAMinusExpr(AMinusExpr node)
{// out of alternative {minus} in Expr, subtract the term
 //  from the expr
   hash.put (node, new Double (getPrim(node.getExpr())
               - getPrim(node.getTerm())));
}

public void outAFactorTerm (AFactorTerm node)
{ //  Value of the term same as the factor
   hash.put (node, getVal(node.getFactor())) ;
}

public void outAMultTerm(AMultTerm node)
{// out of alternative {mult} in Factor, multiply the term
 // by the factor
   hash.put (node, new Double (getPrim(node.getTerm())
               * getPrim(node.getFactor())));
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in Factor, divide the term by
 // the factor
   hash.put (node, new Double (getPrim(node.getTerm())
               / getPrim(node.getFactor())));
}

public void outANumberFactor (ANumberFactor node)
{   hash.put (node, getVal (node.getNumber())); }

public void outAParenFactor (AParenFactor node)
{   hash.put (node, new Double (0.0)); }

double getPrim (Node node)
{  return ((Double) hash.get (node)).doubleValue(); }

Double getVal (Node node)
{  return  hash.get (node) ; }
}
```

6. Show the SableCC grammar which will check for proper syntax of regular expressions over the alphabet {0,1}. Observe the precedence rules for the three operations. Some examples are shown:

```
Valid                    Not Valid

(0+1)*.1.1               *0
0.1.0*                   (0+1)+1)
((0))                    0+
```

5.4 Arrays

Although arrays are not included in our definition of Decaf, they are of such great importance to programming languages and computing in general, that we would be remiss not to mention them at all in a compiler text. We will give a brief description of how multi-dimensional array references can be implemented and converted to atoms, but for a more complete and efficient implementation the student is referred to Parsons [17] or Aho et. al. [1].

The main problem that we need to solve when referencing an array element is that we need to compute an offset from the first element of the array. Though the programmer may be thinking of multi-dimensional arrays (actually arrays of arrays) as existing in two, three, or more dimensions, they must be physically mapped to the computer's memory, which has one dimension. For example, an array declared as `int n[][][] = new int [2][3][4];` might be envisioned by the programmer as a structure having three rows and four columns in each of two planes as shown in Figure 5.10 (a). In reality, this array is mapped into a sequence of twenty-four (2*3*4) contiguous memory locations as shown in Figure 5.10 (b). The problem which the compiler must solve is to convert an array reference such as `n[1][1][0]` to an offset from the beginning of the storage area allocated for n. For this example, the offset would be sixteen memory cells (assuming that each element of the array occupies one memory cell).

To see how this is done, we will begin with a simple one-dimensional array and then proceed to two and three dimensions. For a vector, or one-dimensional array, the offset is simply the subscripting value, since subscripts begin at 0 in Java. For example, if v were declared to contain twenty elements, `char v[] = new char[20];`, then the offset for the fifth element, v[4], would be 4, and in general the offset for a reference v[i] would be i. The simplicity of this formula results from the fact that array indexing begins with 0 rather than 1. A vector maps directly to the computer's memory.

Now we introduce arrays of arrays, which, for the purposes of this discussion, we call multi-dimensional arrays; suppose m is declared as a matrix, or two-dimensional array, `char m[][] = new char [10][15];`. We are thinking of this as an array of 10 rows, with 15 elements in each row. A reference to an element of this array will compute an offset of fifteen elements for each row after

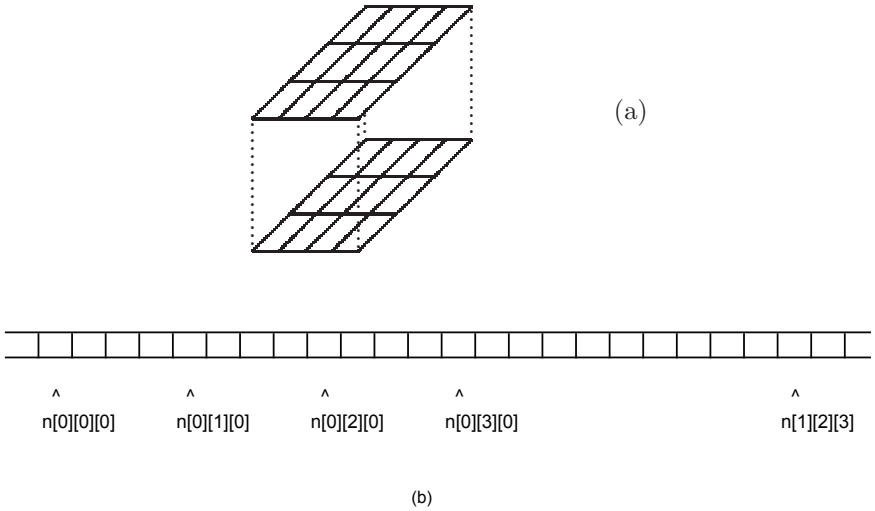


Figure 5.10: A three-dimensional array `n[2][3][4]` (a) Mapped into a one-dimensional memory (b).

the first. Also, we must add to this offset the number of elements in the selected row. For example, a reference to m[4][7] would require an offset of 4*15 + 7 = 67. The reference `m[r][c]` would require an offset of r*15 + c. In general, for a matrix declared as `char m[][] = new char [ROWS][ COLS],` the formula for the offset of `m[r][c]` is r*COLS + c.

For a three-dimensional array, char a[][][] = new char [5][6][7];, we must sum an offset for each plane (6*7 elements), an offset for each row (7 elements), and an offset for the elements in the selected row. For example, the offset for the reference a[2][3][4] is found by the formula 2*6*7 + 3*7 + 4. The reference a[p][r][c] would result in an offset computed by the formula p*6*7 + r*7 + c. In general, for a three-dimensional array, new char [PLANES][ROWS][COLS], the reference a[p][r][c] would require an offset computed by the formula p*ROWS*COLS + r*COLS + c.

We now generalize what we have done to an array that has any number of dimensions. Each subscript is multiplied by the total number of elements in all higher dimensions. If an n dimensional array is declared as `char a[][]...[] = new char[`D_1`][`D_2`][`D_3`]...[`D_n`]`, then a reference to `a[`S_1`][`S_2`][`S_3`]...[`S_n`]` will require an offset computed by the following formula:

S_1*D_2*D_3*D_4*...*D_n + S_2*D_3*D_4*...*D_n + S_3*D_4*...*D_n + ... + S_{n-1}*D_n + S_n.

In this formula, D_i represents the number of elements in the ith dimension and S_i represents the ith subscript in a reference to the array. Note that in some languages, such as Java and C, all the subscripts are not required. For example, the array of three dimensions `a[2][3][4]`, may be referenced with two, one, or even zero subscripts. a[1] refers to the address of the first element in the second

plane; i.e. all missing subscripts are assumed to be zero.

Notice that some parts of the formula shown above can be computed at compile time. For example, for arrays which are dimensioned with constants, the product of dimensions D_2*D_3*D_4 can be computed at compile time. However, since subscripts can be arbitrary expressions, the complete offset may have to be computed at run time.

The atoms which result from an array reference must compute the offset as described above. Specifically, for each dimension, i, we will need a MUL atom to multiply Si by the product of dimensions from D_{i+1} through D_n, and we will need an ADD atom to add the term for this dimension to the sum of the previous terms. Before showing a translation grammar for this purpose, however, we will first show a grammar without action symbols or attributes, which defines array references. Grammar G22 is an extension to the grammar for simple arithmetic expressions, G5, given in section 3.1. Here we have changed rule 7 and added rules 8,9.

G22

1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor → (Expr)
6. Factor → const
7. Factor → var Subs
8. Subs → [Expr] Subs
9. Subs → ϵ

This extension merely states that a variable may be followed by a list of subscripting expressions, each in square brackets (the nonterminal Subs represents a list of subscripts).

Grammar G23 shows rules 7-9 of grammar G22, with attributes and action symbols. Our goal is to come up with a correct offset for a subscripted variable in grammar rule 8, and provide its address for the attribute of the Subs defined in that rule.

Grammar G23:

7. $Factor_e \rightarrow var_v\{MOV\}_{0,,sum} Subs_{v,sum,i}$ $e \leftarrow v[sum]$; $i \leftarrow 1$; $sum \leftarrow Alloc$
8. $Subs_{v,sum,i1} \rightarrow [Expr_e]\{MUL\}_{e,=D,T}\{ADD\}_{sum,T,sum} Subs_{v,sum,i2}$ $D \leftarrow prod(v, i1)$; $i2 \leftarrow i1 + 1$; $T \leftarrow Alloc$
9. $Subs_{v,sum,i} \rightarrow \{check\}_{i,v}$

The nonterminal Subs has three attributes: v (inherited) represents a reference to the symbol table for the array being referenced, sum (synthesized) represents the location storing the sum of the terms which compute the offset, and i (inherited) is the dimension being processed. In the attribute computation rules for grammar rule 8, there is a call to a method prod(v,i). This method computes the product of the dimensions of the array v, above dimension i. As noted above, this product can be computed at compile time. Its value is then stored as a constant, D, and referred to in the grammar as =D.

The first attribute rule for grammar rule 7 specifies e y v[sum]. This means that the value of sum is used as an offset to the address of the variable v, which then becomes the attribute of the Factor defined in rule 7.

The compiler should ensure that the number of subscripts in the array reference is equal to the number of subscripts in the array declaration. If they are not equal, an error message should be put out. This is done by a procedure named check(i,v) which is specified by the action symbol {check}i,v in rule 9. This action symbol represents a procedure call, not an atom. The purpose of the procedure is to compare the number of dimensions of the variable, v, as stored in the symbol table, with the value of i, the number of subscripts plus one. The check(i,v) method simply puts out an error message if the number of subscripts does not equal the number of dimensions, and the parse continues.

To see how this translation grammar works, we take an example of a three-dimensional array declared as int a[][][] = new int[3][5][7]. An attributed derivation tree for the reference a[p][r][c] is shown in Figure 5.11 (for simplicity we show only the part of the tree involving the subscripted variable, not an entire expression). To build this derivation tree, we first build the tree without attributes and then fill in attribute values where possible. Note that the first and third attributes of Subs are inherited and derive values from higher nodes or nodes on the same level in the tree. The final result is the offset stored in the attribute sum, which is added to the attribute of the variable being subscripted to obtain the offset address. This is then the attribute of the Factor which is passed up the tree.

Sample Problem 5.4.1

Assume the array `m` *has been declared to have two planes, four rows, and five columns:* `m = new char[2] [4] [5];`*. Show the attributed derivation tree generated by grammar G23 for the array reference* `m[x][y][z]`*. Use* `Factor` *as the starting nonterminal, and show the subscripting expressions as* `Expr`*, as done in Figure 4.12. Also show the sequence of atoms which would be put out as a result of this array reference.*

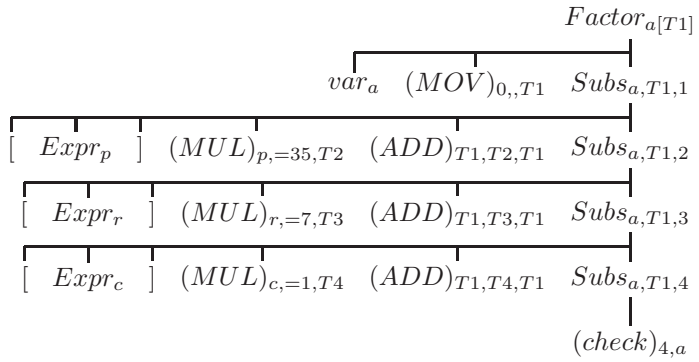
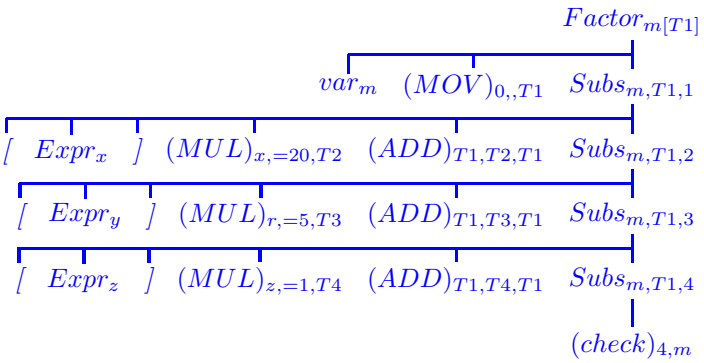


Figure 5.11: A derivation tree for the array reference a[p][r][c], which is declared as int a[3][5][7] using grammar G23.

Solution:



The atoms put out are:

$\{MOV\}_{0,,T1}$ $\{MUL\}_{x,=20,T2}$ $\{ADD\}_{T1,T2,T1}$ $\{MUL\}_{y,=5,T3}$ $\{ADD\}_{T1,T3,T1}$ $\{MUL\}_{z,=1,T4}$ $\{ADD\}_{T1,T4,T1}$ $\{check\}_{4,m}$

5.4.1 Exercises

1. Assume the following array declarations:

```
int v[] = new int [13];
```

```
int m[][] = new int [12][17];
int a3[][][] = new int [15][7][5];
int z[][][][] = new int [4][7][2][3];
```

Show the attributed derivation tree resulting from grammar G23 for each of the following array references. Use Factor as the starting nonterminal, and show each subscript expression as Expr, as done in Figure 5.11. Also show the sequence of atoms that would be put out.

(a) v[7]

(b) m[q][2]

(c) a3[11][b][4]

(d) z[2][c][d][2]

(e) m[1][1]

2. The discussion in this section assumed that each array element occupied one addressable memory cell. If each array element occupies SIZE memory cells, what changes would have to be made to the general formula given in this section for the offset? How would this affect grammar G23?

3. 3. You are given two vectors: the first, d, contains the dimensions of a declared array, and the second, s, contains the subscripting values in a reference to that array.

   (a) Write a Java method :

   ```
   int offSet (int d[], int s[]);
   ```

   that computes the offset for an array reference a$[s_0][s_1]...[s_{max-1}]$ where the array has been declared as char a$[d_0][d_1]$ ... $[dmax - 1]$.

   (b) Improve your Java method, if possible, to minimize the number of run-time multiplications.

5.5 Case Study: Syntax Analysis for Decaf

In this section we continue the development of a compiler for Decaf, a small subset of the Java programming language. We do this by implementing the syntax analysis phase of the compiler using SableCC as described in Section 5.3, above. The parser generated by SableCC will obtain input tokens from the standard input stream. The parser will then check the tokens for correct syntax.

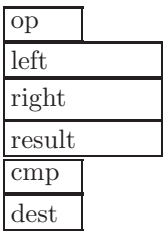
In addition, we provide a Translation class which enables our parser to put out atoms corresponding to the run-time operations to be performed. This aspect of compilation is often called semantic analysis. For more complex languages, semantic analysis would also involve type checking, type conversions, identifier scopes, array references, and symbol table management. Since these

will not be necessary for the Decaf compiler, syntax analysis and semantic analysis have been combined into one program.

The complete SableCC grammar file and Translation source code is shown in AppendixB and is explained here. The input to SableCC is the file decaf.grammar, which generates classes for the parser, nodes, lexer, and analysis. In the Tokens section, we define the two types of comments; comment1 is a single-line comment, beginning with // and ending with a newline character. comment2 is a multi-line comment, beginning with /* and ending with */. Neither of these tokens requires the use of states, which is why there is no States section in our grammar. Next each keyword is defined as a separate token taking care to include these before the definition of identifiers. These are followed by special characters '+', '-', ;, ... Note that relational operators are defined collectively as a compare token. Finally we define identifiers and numeric constants as tokens. The Ignored Tokens are space and the two comment tokens.

The Productions section is really the Decaf grammar with some modifications to allow for bottom-up parsing. The major departure from what has been given previously and in Appendix A, is the definition of the if statement. We need to be sure to handle the dangling else appropriately; this is the ambiguity problem discussed in section 3.1 caused by the fact that an `if` statement has an optional `else` part. This problem was relatively easy to solve when parsing top-down, because the ambiguity was always resolved in the correct way simply by checking for an else token in the input stream. When parsing bottom-up, however, we get a shift-reduce conflict from this construct. If we rewrite the grammar to eliminate the ambiguity, as in section 3.1 (Grammar G7), we still get a shift-reduce conflict. Unfortunately, in SableCC there is no way to resolve this conflict always in favor of a shift (this is possible with yacc). Therefore, we will need to rewrite the grammar once again; we use a grammar adapted from Appel [3]. In this grammar a no_short_if statement is one which does not contain an if statement without a matching else. The EBNF capabilities of SableCC are used, for example, in the definition of compound_stmt, which consists of a pair of braces enclosing 0 or more statements. The complete grammar is shown in appendix B. An array of Doubles named 'memory' is used to store the values of numeric constants.

The Translation class, also shown in appendix B, is written to produce atoms for the arithmetic operations and control structures. The structure of an atom is shown in Figure 5.12. The Translation class uses a few Java maps: the first map, implemented as a HashMap and called 'hash', stores the temporary memory location associated with each sub-expression (i.e. with each node in the syntax tree). It also stores label numbers for the implementation of control structures. Hence, the keys for this map are nodes, and the values are the integer run-time memory locations, or label numbers, associated with them. The second map, called 'nums', stores the values of numeric constants, hence if a number occurs several times in a Decaf program, it need be stored only once in this map. The third map is called 'identifiers'. This is our Decaf symbol table. Each identifier is stored once, when it is declared. The Translation class checks that an identifier is not declared more than once (local scope is not permitted), and it checks that an



Operation of Atom
Left operand location
Right operand location
Result operand location
Comparison code for TST atoms
Destination, for JMP, LBL, and TST atoms

Figure 5.12: Record structure of the file of atoms

identifier has been declared before it is used. For both numbers and identifiers, the value part of each entry stores the run-time memory location associated with it. The implementation of control structures for if, while, and for statements follows that which was presented in section 4.9. A boolean expression always results in a TST atom which branches if the comparison operation result is false. Whenever a new temporary location is needed, the method alloc provides the next available location (a better compiler would re-use previously allocated locations when possible). Whenever a new label number is needed, it is provided by the lalloc method. Note that when an integer value is stored in a map, it must be an object, not a primitive. Therefore, we use the wrapper class for integers provided by Java, Integer. The complete Translation class is shown in appendix B and is available at `http://www.rowan.edu/~bergmann/books`.

For more documentation on SableCC, visit `http://www.sablecc.org`.

5.5.1 Exercises

1. Extend the Decaf language to include a do statement defined as:

   DoStmt → do Stmt while (BoolExpr) ;

   Modify the files decaf.grammar and Translation.java, shown in Appendix B so that the compiler puts out the correct atom sequence implementing this control structure, in which the test for termmination is made after the body of the loop is executed. The nonterminals Stmt and BoolExpr are already defined. For purposes of this assignment you may alter the atom method so that it prints out its arguments to stdout rather than building a file of atoms.

2. Extend the Decaf language to include a switch statement defined as:

   SwitchStmt → switch (Expr) CaseList
   CaseList → case number ':' Stmt CaseList
   CaseList → case number ':' Stmt

   Modify the files decaf.grammar and Translation.java, shown in Appendix B, so that the compiler puts out the correct atom sequence implement-

ing this control structure. The nonterminals Expr and Stmt are already defined, as are the tokens number and end. The token switch needs to be defined. Also define a break statement which will be used to transfer control out of the switch statement. For purposes of this assignment, you may alter the atom() function so that it prints out its arguments to stdout rather than building a file of atoms, and remove the call to the code generator.

3. Extend the Decaf language to include initializations in decalarations, such as:

   int x=3, y, z=0;

   Modify the files decaf.grammar and Translation.java, shown in Appendix B, so that the compiler puts out the correct atom sequence implementing this feature. You will need to put out a MOV atom to assign the value of the constant to the variable.

5.6 Chapter Summary

This chapter describes some bottom up parsing algorithms. These algorithms recognize a sequence of grammar rules in a derivation, corresponding to an upward direction in the derivation tree. In general, these algorithms begin with an empty stack, read input symbols, and apply grammar rules, until left with the starting nonterminal alone on the stack when all input symbols have been read.

The most general class of bottom up parsing algorithms is called shift reduce parsing. These parsers have two basic operations: (1) a shift operation pushes the current input symbol onto the stack, and (2) a reduce operation replaces zero or more top-most stack symbols with a single stack symbol. A reduction can be done only if a handle can be identified on the stack. A handle is a string of symbols occurring on the right side of a grammar rule, and matching the symbols on top of the stack, as shown below:

$\bigtriangledown$ ... HANDLE Nt $\rightarrow$ HANDLE

The reduce operation applies the rewriting rule in reverse, by replacing the handle on the stack with the nonterminal defined in the corresponding rule, as shown below

$\bigtriangledown$ ... Nt

When writing the grammar for a shift reduce parser, one must take care to avoid shift/reduce conflicts (in which it is possible to do a reduce operation when a shift is needed for a correct parse) and reduce/reduce conflicts (in which more than one grammar rule matches a handle).

A special case of shift reduce parsing, called LR parsing, is implemented with a pair of tables: an action table and a goto table. The action table specifies whether a shift or reduce operation is to be applied. The goto table specifies the stack symbol to be pushed when the operation is a reduce.

We studied a parser generator, SableCC, which generates an LR parser from a specification grammar. It is also possible to include actions in the grammar which are to be applied as the input is parsed. These actions are implemented in a Translation class designed to be used with SableCC.

Finally we looked at an implementation of Decaf, our case study language which is a subset of Java, using SableCC. This compiler works with the lexical phase discussed in section 2.4 and is shown in Appendix B.

Chapter 6

Code Generation

6.1 Introduction to Code Generation

Up to this point we have ignored the architecture of the machine for which we are building the compiler, i.e. the target machine. By architecture, we mean the definition of the computer's central processing unit as seen by a machine language programmer. Specifications of instruction-set operations, instruction formats, addressing modes, data formats, CPU registers, input/output instructions, etc. all make up what is sometime called the conventional machine language architecture (to distinguish it from the microprogramming level architecture which many computers have; see, for example, Tanenbaum [21]). Once these are all clearly and precisely defined, we can complete the compiler by implementing the code generation phase. This is the phase which accepts as input the syntax trees or stream of atoms as put out by the syntax phase, and produces, as output, the object language program in binary coded instructions in the proper format.

The primary objective of the code generator is to convert atoms or syntax trees to instructions. In the process, it is also necessary to handle register allocation for machines that have several general purpose CPU registers. Label atoms must be converted to memory addresses. For some languages, the compiler has to check data types and call the appropriate type conversion routines if the programmer has mixed data types in an expression or assignment.

Note that if we are developing a new computer, we do not need a working model of that computer in order to complete the compiler; all we need are the specifications, or architecture, of that computer. Many designers view the construction of compilers as made up of two logical parts - the front end and the back end. The front end consists of lexical and syntax analysis and is machine-independent. The back end consists of code generation and optimization and is very machine-dependent, consequently this chapter commences our discussion of the back end, or machine-dependendent, phases of the compiler.

This separation into front and back ends simplifies things in two ways when constructing compilers for new machines or new languages. First, if we are

implementing a compiler for a new machine, and we already have compilers for our old machine, all we need to do is write the back end, since the front end is not machine dependent. For example, if we have a Pascal compiler for an IBM PS/2, and we wish to implement Pascal on a new RISC (Reduced Intruction Set Computer) machine, we can use the front end of the existing Pascal compiler (it would have to be recompiled to run on the RISC machine). This means that we need to write only the back end of the new compiler (refer to Figure 1.9).

Our life is also simplified when constructing a compiler for a new programming language on an existing computer. In this case, we can make use of the back end already written for our existing compiler. All we need to do is rewrite the front end for the new language, compile it, and link it together with the existing back end to form a complete compiler. Alternatively, we could use an editor to combine the source code of our new front end with the source code of the back end of the existing compiler, and compile it all at once.

For example, suppose we have a Pascal compiler for the Macintosh, and we wish to construct an Ada compiler for the Macintosh. First, we understand that the front end of each compiler translates source code to a string of atoms (call this language Atoms), and the back end translates Atoms to Mac machine language (Motorola 680x0 instructions). The compilers we have are $C_{\text{Pas}}^{\text{Pas} \rightarrow \text{Mac}}$ and $C_{\text{Mac}}^{\text{Pas} \rightarrow \text{Mac}}$, the compiler we want is $C_{\text{Mac}}^{\text{Ada} \rightarrow \text{Mac}}$, and each is composed of two parts, as shown in Figure 6.1. We write $C_{\text{Pas}}^{\text{Ada} \rightarrow \text{Atoms}}$, which is the front end of an Ada compiler and is also shown in Figure 6.1.

We then compile the front end of our Ada compiler as shown in Figure 6.2 and link it with the back end of our Pascal compiler to form a complete Ada compiler for the Mac, as shown in Figure 6.3.

The back end of the compiler consists of the code generation phase, which we will discuss in this chapter, and the optimization phases, which will be discussed in Chapter 7. Code generation is probably the least intensively studied phase of the compiler. Much of it is straightforward and simple; there is no need for extensive research in this area. In the past most of the research that has been done is concerned with methods for specifying target machine architectures, so that this phase of the compiler can be produced automatically, as in a compiler-compiler. In more recent years, research has centered on generating code for embedded systems, special-purpose computers, and multi-core systems.

Sample Problem 6.1.1

Assume we have a Pascal compiler for a Mac (both source and executable code) as shown in Figure 6.1. We are constructing a completely new machine called a RISC, for which we wish to construct a Pascal compiler. Show how this can be done without writing the entire compiler and without writing any machine or assembly language

We have the source code for a Pascal compiler:

$$C^{\text{Pas} \to \text{Mac}}_{\text{Pas}} = C^{\text{Pas} \to \text{Atoms}}_{\text{Pas}} + C^{\text{Atoms} \to \text{Mac}}_{\text{Pas}}$$

We have the Pascal compiler which runs on the Mac:

$$C^{\text{Pas} \to \text{Mac}}_{\text{Mac}} = C^{\text{Pas} \to \text{Atoms}}_{\text{Mac}} + C^{\text{Atoms} \to \text{Mac}}_{\text{Mac}}$$

We want an Ada compiler which runs on the Mac:

$$C^{\text{Ada} \to \text{Mac}}_{\text{Mac}} = C^{\text{Ada} \to \text{Atoms}}_{\text{Mac}} + C^{\text{Atoms} \to \text{Mac}}_{\text{Mac}}$$

We write the front end of the Ada compiler in Pascal:

$$C^{\text{Ada} \to \text{Atoms}}_{\text{Pas}}$$

Figure 6.1: Using a Pascal compiler to construct an Ada compiler

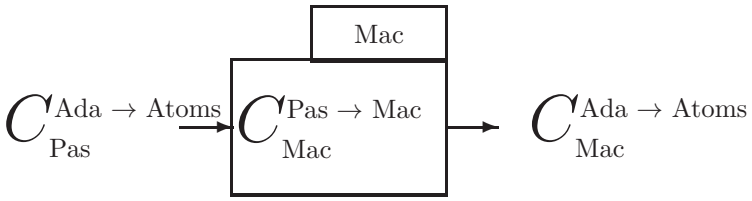


Figure 6.2: Compile the front end of the Ada compiler on the Mac

$$C^{\text{Ada} \to \text{Atoms}}_{\text{Mac}} + C^{\text{Atoms} \to \text{Mac}}_{\text{Mac}} = C^{\text{Ada} \to \text{Mac}}_{\text{Mac}}$$

Figure 6.3: Link the front end of the Ada compiler with the back end of the Pascal compiler to produce a complete Ada compiler.

code.

Solution:

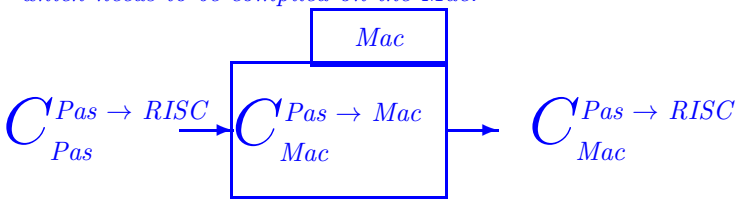
We want $C^{Ada \rightarrow RISC}_{RISC}$

Write (in Pascal) the back end of a compiler for the RISC machine:

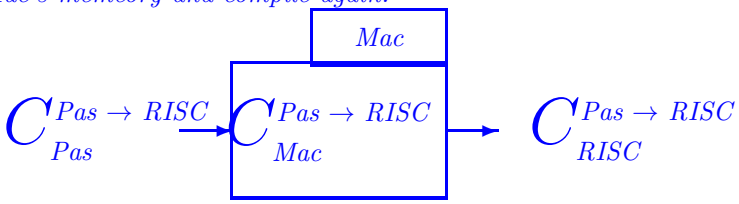
$C^{Atoms \rightarrow RISC}_{Pas}$

We now have $C^{Pas \rightarrow RISC}_{Pas} = C^{Pas \rightarrow Atoms}_{Pas} + C^{Atoms \rightarrow RISC}_{Pas}$

which needs to be compiled on the Mac:



But this is still not what we want, so we load the output into the Mac's memeory and compile again:



and the output is the compiler that we wanted to generate.

6.1.1 Exercises

1. Show the big C notation for each of the following compilers (assume that each uses an intermediate form called *Atoms*):

   (a) The back end of a compiler for the Sun computer.

(b) The source code, in Pascal, for a COBOL compiler whose target machine is the PC.

(c) The souce code, in Pascal, for the back end of a FORTRAN compiler for the Sun.

2. Show how to generate $\mathcal{C}^{\text{Lisp} \to \text{PC}}_{\text{PC}}$ without writing any more programs, given a PC machine and each of the following collections of compilers:

   (a) $\mathcal{C}^{\text{Lisp} \to \text{PC}}_{\text{Pas}}$ $\mathcal{C}^{\text{Pas} \to \text{PC}}_{\text{PC}}$

   (b) $\mathcal{C}^{\text{Lisp} \to \text{Atoms}}_{\text{Pas}}$ $\mathcal{C}^{\text{Pas} \to \text{Atoms}}_{\text{Pas}}$ $\mathcal{C}^{\text{Atoms} \to \text{PC}}_{\text{Pas}}$ $\mathcal{C}^{\text{Pas} \to \text{PC}}_{\text{PC}}$

   (c) $\mathcal{C}^{\text{Lisp} \to \text{Atoms}}_{\text{PC}}$ $\mathcal{C}^{\text{Atoms} \to \text{PC}}_{\text{PC}}$

3. Given a Sparc computer and the following compilers, show how to generate a Pascal (Pas) compiler for the MIPS machine without doing any more programming. (Unfortunately, you cannot afford to buy a MIPS computer.)

$$\mathcal{C}^{\text{Pas} \to \text{Sparc}}_{\text{Pas}} = \mathcal{C}^{\text{Pas} \to \text{Atoms}}_{\text{Pas}} + \mathcal{C}^{\text{Atoms} \to \text{Sparc}}_{\text{Pas}}$$

$$\mathcal{C}^{\text{Pas} \to \text{Sparc}}_{\text{Sparc}} = \mathcal{C}^{\text{Pas} \to \text{Atoms}}_{\text{Sparc}} + \mathcal{C}^{\text{Atoms} \to \text{Sparc}}_{\text{Sparc}}$$

$$\mathcal{C}^{\text{Atoms} \to \text{MIPS}}_{\text{Pas}}$$

6.2 Converting Atoms to Instructions

If we temporarily ignore the problem of forward references (of Jump or Branch instructions), the process of converting atoms to instructions is relatively simple. For the most part all we need is some sort of case, switch, or multiple destination branch based on the class of the atom being translated. Each atom class would result in a different instruction or sequence of instructions. If the CPU of the target machine requires that all arithmetic be done in registers, then an example of a translation of an ADD atom would be as shown in Figure 6.4; i.e., an ADD atom is translated into a LOD (Load Into Register) instruction, followed by an ADD instruction, followed by a STO (Store Register To Memory) instruction.

```
(ADD, a, b, T1)    →    LOD r1,a
                        ADD r1,b
                        STO r1,T1
```

Figure 6.4: Translation of an ADD atom to instructions

Most of the atom classes would be implemented in a similar way. Conditional Branch atoms (called TST atoms in our examples) would normally be implemented as a Load, Compare, and Branch, depending on the architecture of the target machine. The MOV (move data from one memory location to another) atom could be implemented as a MOV (Move) instruction, if permitted by the target machine architecture; otherwise it would be implemented as a Load followed by a Store.

Operand addresses which appear in atoms must be appropriately coded in the target machine's instruction format. For example, many target machines require operands to be addressed with a base register and an offset from the contents of the base register. If this is the case, the code generator must be aware of the presumed contents of the base register, and compute the offset so as to produce the desired operand address. For example, if we know that a particular operand is at memory location 1E (hex), and the contents of the base register is 10 (hex), then the offset would be 0E, because 10 + 0E = 1E. In other words, the contents of the base register, when added to the offset, must equal the operand address.

Sample Problem 6.2.1

---

*The Java statement if (a>b) a = b * c; might result in the following sequence of atoms:*

```
(TST, A, B,, 4, L1)          // Branch to L1 if A<=B
(MUL, B, C, A)
(LBL L1)
```

Translate these atoms to instructions for a Load/Store architecture. Assume that the operations are LOD (Load), STO (Store), ADD, SUB, MUL, DIV, CMP (Compare), and JMP (Conditional Branch). The Compare instruction will set a flag for the Jump instruction, and a comparison code of 0 always sets the flag to True, which results in an Unconditional Branch. Assume that variables and labels may be represented by symbolic addresses.

Solution:

```
        LOD     r1,a            // Load a into reg. r1
        CMP     r1,b,4          // Compare a <= B?
        JMP     L1              // Branch if true
        LOD     r1,b
        MUL     r1,c            // r1 = b * c
        STO     r1,a            // a = b * c
L1:
```

6.2.1 Exercises

1. For each of the following Java statements we show the atom string produced by the parser. Translate each atom string to instructions, as in the sample problem for this section. You may assume that variables and labels are represented by symbolic addresses.

   (a)
   ```
   {       a = b + c * (d - e) ;
           b = a;
   }
   ```

   ```
   (SUB, d, e, T1)
   (MUL, c, T1, T2)
   (ADD, b, T2, T3)
   (MOV, T3,, a)
   (MOV, a,, b)
   ```

   (b)
   ```
           for (i=1; i<=10; i++) j = j/3 ;
   ```

   ```
   (MOV, 1,, i)
   (LBL, L1)
   (TST, i, 10,, 3, L4)                // Branch if i>10
   (JMP, L3)
   (LBL, L5)
   (ADD, 1, i, i)                      // i++
   ```

```
(JMP, L1)                          // Repeat the loop
(LBL, L3)
(DIV, j, 3, T2)                    // T2 = j / 3;
(MOV, T2,, j)                      // j = T2;
(JMP, L5)
(LBL, L4)                          // End of loop
```

(c)

```
        if (a!=b+3) a = 0; else b = b+3;

(ADD, b, 3, T1)
(TST, a, T1,, 1, L1)           // Branch if a==b+3
(MOV, 0,, a)                   // a = 0
(JMP, L2)
(LBL, L1)
(ADD, b, 3, T2)                // T2 = b + 3
(MOV, T2,, b)                  // b = T2
(LBL, L2)
```

2. How many instructions correspond to each of the following atom classes on a Load/Store architecture, as in the sample problem of this section?

(a)	ADD	(b)	DIV	(c)	MOV
(d)	TST	(e)	JMP	(f)	LBL

3. Why is it important for the code generator to know how many instructions correspond to each atom class?

4. How many machine language instructions would correspond to an ADD atom on each of the following architectures?

   (a) Zero address architecture (a stack machine)

   (b) One address architecture

   (c) Two address architecture

   (d) Three address architecture

6.3 Single Pass vs. Multiple Passes

There are several different ways of approaching the design of the code generation phase. The difference between these approaches is generally characterized by the number of passes which are made over the input file. For simplicity, we will assume that the input file is a file of atoms, as specified in Chapters 4 and

Atom	Location	Instruction
(ADD, a, b, T1)	4	LOD r1,a
	5	ADD 41,b
	6	STO r1,T1
(JMP, L1)	7	CMP 0,0,0
	8	JMP ?
(LBL, L1)		(L1 = 9)

Figure 6.5: Problem in generating a jump to a forward destination

5. A code generator which scans this file of atoms once is called a single pass code generator, and a code generator which scans it more than once is called a multiple pass code generator.

The most significant problem relevant to deciding whether to use a single or multiple pass code generator has to do with forward jumps. As atoms are encountered, instructions can be generated, and the code generator maintains a memory address counter, or program counter. When a Label atom is encountered, a memory address value can be assigned to that Label atom (a table of labels is maintained, with a memory address assigned to each label as it is defined). If a Jump atom is encountered with a destination that is a higher memory address than the Jump instruction (i.e. a forward jump), the label to which it is jumping has not yet been encountered, and it will not be possible to generate the Jump instruction completely at this time. An example of this situation is shown in Figure 6.5 in which the jump to Label L1 cannot be generated because at the time the JMP atom is encountered the code generator has not encountered the definition of the Label L1, which will have the value 9.

A JMP atom results in a CMP (Compare instruction) followed by a JMP (Jump instruction), to be consistent with the sample architecture presented in section 6.5, below. There are two fundamental ways to resolve the problem of forward jumps. Single pass compilers resolve it by keeping a table of Jump instructions which have forward destinations. Each Jump instruction with a forward reference is generated incompletely (i.e., without a destination address) when encountered, and each is also entered into a fixup table, along with the Label to which it is jumping. As each Label definition is encountered, it is entered into a table of Labels, along with its address value. When all of the atoms have been read, all of the Label atoms will have been defined, and, at this time, the code generator can revisit all of the Jump instructions in the Fixup table and fill in their destination addresses. This is shown in Figure 6.6 for the same atom sequence shown in Figure 6.5. Note that when the (JMP, L1) atom is encountered, the Label L1 has not yet been defined, so the location of the Jump (8) is entered into the Fixup table. When the (LBL, L1) atom is encountered, it is entered into the Label table, because the target machine address corresponding to this Label (9) is now known. When the end of file (EOF) is encountered, the destination of the Jump instruction at location 8 is

			Fixup Table		Label Table	
Atom	Loc	Instruction	Loc	Label	Label	Value
(ADD,a,b,T1)	4	LOD r1,a				
	5	ADD r1,b				
	6	STO r1,T1				
(JMP,L1)	7	CMP 0,0,0				
	8	JMP 0	8	L1		
(LBL,L1)					L1	9
...						
EOF						
	8	JMP 9				

Figure 6.6: Use of the Fixup Table and Label Table in a single pass code generator

changed, using the Fixup table and the Label table, to 9.

Multiple pass code generators do not require a Fixup table. In this case, the first pass of the code generator does nothing but build the table of Labels, storing a memory address for each Label. Then, in the second pass, all the Labels will have been defined, and each time a Jump is encountered its destination Label will be in the table, with an assigned memory address. This method is shown in Figure 6.7 which, again, uses the atom sequence given in Figure 6.5.

Note that, in the first pass, the code generator needs to know how many machine language instructions correspond to an atom (three to an ADD atom and two to a JMP atom), though it does not actually generate the instructions. It can then assign a memory address to each Label in the Label table.

A single pass code generator could be implemented as a subroutine to the parser. Each time the parser generates an atom, it would call the code generator to convert the atom to machine language and put out the instruction(s) corresponding to that atom. A multiple pass code generator would have to read from a file of atoms, created by the parser, and this is the method we use in our sample code generator in section 6.5.

Sample Problem 6.3.1

The following atom string resulted from the Java statement
`while (i<=x) { x = x+2; i = i*3; }`
Translate it into instructions as in (1) a single pass code generator using a Fixup table and (2) a multiple pass code generator.

```
(LBL, L1)
(TST, i, x,, 3, L2)            // Branch if T1 is false
(ADD, x, 2, T1)
```

Begin First Pass:

			Label Table	
Atom	Loc	Instruction	Label	Value
(ADD,a,b,T1)	4-6			
(JMP,L1)	7-8			
(LBL,L1)			L1	9
...				
EOF				

Begin Second Pass:

Atom	Loc	Instruction
(ADD,a,b,T1)	4	LOD r1,a
	5	ADD r1,b
	6	STO r1,T1
(JMP,L1)	7	CMP 0,0,0
	8	JMP 9
(LBL,L1)		
...		
EOF		

Figure 6.7: Forward jumps handled by a multiple pass code generator

```
(MOV, T1, , x)
(MUL, i, 3, T2)
(MOV, T2, , i)
(JMP, L1)                          // Repeat the loop
(LBL, L2)                          // End of loop
```

Solution:

(1) Single Pass

```
                                                  Fixup Table          Label Table
Atom              Loc Instruction                 Loc      Label       Label   Value
(LBL, L1)         0                                                    L1      0
(TST,i,x,,3,L2)   0   CMP  i,x,3
                  1   JMP  ?                     1        L2
(ADD, X, 2, T1)   2   LOD  R1,x
                  3   ADD  R1,2
                  4   STO  R1,T1
(MOV, T1,, x)     5   LOD  R1,T1
                  6   STO  R1,x
(MUL, i, 3, T2)   7   LOD  R1,i
                  8   MUL  R1,3
                  9   STO  R1,T2
(MOV, T2,, i)     10  LOD  R1,T2
                  11  STO  R1,i
(JMP, L1)         12  CMP  0,0,0
                  13  JMP  0
(LBL, L2)         14                                                   L2      14

...
                  1   JMP  14
```

(2) Multiple passes

Begin First Pass:

```
                                                  Label Table
Atom                Loc      Instruction          Label   Value
(LBL, L1)           0                             L1      0
(TST,i,x,,3,L2)     0
(ADD, X, 2, T1)     2
```

```
(MOV, T1,, x)        5
(MUL, i, 3, T2)      7
(MOV, T2,, i)        10
(JMP, L1)            12
(LBL, L2)            14                              L2      14

...
```

Begin Second Pass:

```
Atom                 Loc   Instruction
(LBL, L1)            0
(TST,i,x,,3,L2)      0     CMP  i,x,3
                     1     JMP  14
(ADD, X, 2, T1)      2     LOD  R1,x
                     3     ADD  R1,2
                     4     STO  R1,T1
(MOV, T1,, x)        5     LOD  R1,T1
                     6     STO  R1,x
(MUL, i, 3, T2)      7     LOD  R1,i
                     8     MUL  R1,3
                     9     STO  R1,T2
(MOV, T2,, i)        10    LOD  R1,T2
                     11    STO  R1,i
(JMP, L1)            12    CMP  0,0,0
                     13    JMP  0
(LBL, L2)            14
```

6.3.1 Exercises

1. The following atom string resulted from the Java statement:

   ```
   for (i=a; i<b+c; i++) b = b/2;
   ```

   Translate the atoms to instructions as in the sample problem for this section using two methods: (1) a single pass method with a Fixup table for forward Jumps and (2) a multiple pass method. Refer to the variables a,b,c symbolically.

```
(MOV, a,, i)
(LBL, L1)
(ADD, b, c, T1)            // T1 = b+c
(TST, i, T1,, 4, L2)       // If i>=b+c, exit loop
(JMP, L3)                  // Exit loop
(LBL, L4)
(ADD, i, 1, i)             // Increment i
(JMP, L1)                  // Repeat loop
(LBL, L3)
(DIV, b, ='2', T3)         // Loop Body
(MOV, T3,, b)
(JMP, L4)                  // Jump to increment
(LBL, L2)
```

2. Repeat Problem 1 for the atom string resulting from the Java statement:

```
if (a==(b-33)*2) a = (b-33)*2;
   else a = x+y;

(SUB, b, ='33', T1)
(MUL, T1, ='2', T2)            // T2 = (b-33)*2
(TST, a, T2,, 6, L1)           // Branch if a!=T2
(SUB, b, ='33', T3)
(MUL, T3, ='2', T4)
(MOV, T4,, a)
(JMP, L2)                      // Skip else part
(LBL, L1)                      // else part
(ADD, x, y, T5)
(MOV, T5,, a)
(LBL, L2)
```

3. (a) What are the advantages of a single pass method of code generation over a multiple pass method?

   (b) What are the advantages of a multiple pass method of code generation over a single pass method?

6.4 Register Allocation

Some computers (such as the DEC PDP-8) are designed with a single arithmetic register, called an accumulator, in which all arithmetic operations are performed. Other computers (such as the Intel 8086) have only a few CPU registers, and

they are not general purpose registers; i.e., each one has a limited range of uses or functions. In these cases the allocation of registers is not a problem.

However, most modern architectures have many CPU registers; the DEC Vax, IBM mainframe, MIPS, and Motorola 680x0 architectures each has 16-32 general purpose registers, for example, and the RISC (Reduced Instruction Set Computer) architectures, such as the SUN SPARC, generally have about 500 CPU registers (though only 32 are used at a time). In this case, register allocation becomes an important problem. Register allocation is the process of assigning a purpose to a particular register, or binding a register to a programmer variable or compiler variable, so that for a certain range or scope of instructions that register has the specified purpose or binding and is used for no other purposes. The code generator must maintain information on which registers are used for which purposes, and which registers are available for reuse. The main objective in register allocation is to maximize utilization of the CPU registers, and to minimize references to memory locations.

It might seem that register allocation is more properly a topic in the area of code optimization, since code generation could be done with the assumption that there is only one CPU register (resulting in rather inefficient code). Nevertheless, register allocation is always handled (though perhaps not in an optimal way) in the code generation phase. A well chosen register allocation scheme can not only reduce the number of instructions required, but it can also reduce the number of memory references. Since operands which are used repeatedly can be kept in registers, the operands do not need to be recomputed, nor do they need to be loaded from memory. It is especially important to minimize memory references in compilers for RISC machines, in which the objective is to execute one instruction per machine cycle, as described in Tanenbaum [21].

An example, showing the importance of smart register allocation, is shown in Figure 6.8 for the two statement program segment:

```
a = b + c * d ;
a = a - c * d ;
```

The smart register allocation scheme takes advantage of the fact that C*D is a common subexpression, and that the variable A is bound, temporarily, to register R2. If no attention is paid to register allocation, the two statements in Figure 6.8 are translated into twelve instructions, involving a total of twelve memory references. With smart register allocation, however, the two statements are translated into seven instructions, with only five memory references. (Some computers, such as the VAX, permit arithmetic on memory operands, in which case register allocation takes on lesser importance.)

An algorithm which takes advantage of repeated subexpressions will be discussed in Section 7.2. Here, we will discuss an algorithm which determines how many registers will be needed to evaluate an expression without storing subexpressions to temporary memory locations. This algorithm will also determine the sequence in which subexpressions should be evaluated to minimize register usage.

This register allocation algorithm will require a syntax tree for an expression to be evaluated. Each node of the syntax tree will have a weight associated

Simple Register Allocation	Smart Register Allocation	
LOD r1,c	LOD r1,c	
MUL r1,d	MUL r1,d	c*d
STO r1,Temp1	LOD r2,B	
LOD r1,b	ADD r2,r1	b+c*d
ADD r1,Temp1	STO r2,a	
STO r1,a	SUB r2,r1	a-c*d
LOD r1,c	STO r2,b	a-c*d
MUL r1,d		
STO r1,Temp2		
LOD r1,a		
SUB r1,Temp2		
STO r1,b		

Figure 6.8: Register allocation, simple and smart, for a two statement program: `a = b+c*d; a = b-c*d;`

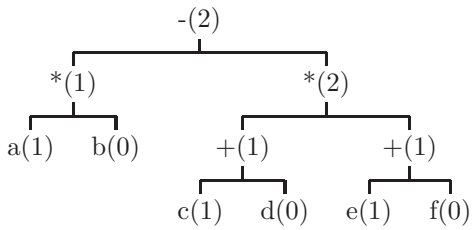


Figure 6.9: A weighted syntax tree for `a*b-(c+d)*(e+f)` with weights shown in parentheses

with it which tells us how many registers will be needed to evaluate each subexpression without storing to temporary memory locations. Each leaf node which is a left operand will have a weight of one, and each leaf node which is a right operand will have a weight of zero. The weight of each interior node will be computed from the weights of its two children as follows: If the two children have different weights, the parent's weight is the maximum of the two children. If the two children have the same weight, w, then the parent's weight is w+1. As an example, the weighted syntax tree for the expression a*b - (c+d) * (e+f) is shown in Figure 6.9 from which we can see that the entire expression should require two registers.

Intuitively, if two expressions representing the two children of a node, N, in a syntax tree require the same number of registers, we will need an additional register to store the result for node N, regardless of which subexpression is evaluated first. In the other case, if the two subexpressions do not require the same number of registers, we can evaluate the one requiring more registers first,

```
LOD     r1,c
ADD     r1,d          r1 = c + d
LOD     r2,3
ADD     r2,f          r2 = e + f
MUL     r1,r2         r1 = (c+d) * (e+f)
LOD     r2,a
MUL     r2,b          r2 = a * b
SUB     r2,41         r2 = a*b - (c+d)*(e+f)
```

Figure 6.10: Code generated for `a*b-(c+d) * (e+f)` using Figure 6.9

at which point those registers are freed for other use.

We can now generate code for this expression. We do this by evaluating the operand having greater weight, first. If both operands of an operation have the same weight, we evaluate the left operand first. For our example in Figure 6.9 we generate the code shown in Figure 6.10. We assume that there are register-register instructions (i.e., instructions in which both operands are contained in registers) for the arithmetic operations in the target machine architecture. Note that if we had evaluated a*b first we would have needed either an additional register or memory references to a temporary location.

This problem would have had a more interesting solution if the expression had been e+f - (c+d)*(e+f) because of the repeated subexpression e+f. If the value of e+f were left in a register, it would not have to be recomputed. There are algorithms which handle this kind of problem, but they will be covered in the chapter on optimization (Chapter 7).

Sample Problem 6.4.1

---

*Use the register allocation algorithm of this section to show a weighted syntax tree for the expression a - b/c + d * (e-f + g*h), and show the resulting instructions, as in Figure 6.10.*

Solution:

```
LOD     r1,a
LOD     r2,b
DIV     r2,c          b/c
```

```
SUB     r1,r2          a - b/c
LOD     r2,e
SUB     r2,f           e - f
LOD     r3,g
MUL     r3,h           g * h
ADD     r2,r3          e - f + g * h
LOD     r3,d
MUL     r3,r2          d * (e-f + g*h)
ADD     r1,r3          a - b/c + d * (e-f + g*h)
```

6.4.1 Exercises

1. Use the register allocation algorithm given in this section to construct a weighted syntax tree and generate code for each of the given expressions, as done in Sample Problem **??**. Do not attempt to optimize for common subexpressions.

   (a) a + b * c - d

   (b) a + (b + (c + (d + e)))

   (c) (a + b) * (c + d) - (a + b) * (c + d)

   (d) a / (b + c) - (d + (e - f)) + (g - h * i) * (j * (k / m))

2. Show an expression different in structure from those in Problem 1 which requires:

   (a) two registers (b) three registers

   As in Problem 1, assume that common subexpressions are not detected and that Loads and Stores are minimized.

3. Show how the code generated in Problem 1 (c) can be improved by making use of common subexpressions.

6.5 Case Study: A Code Generator for the Mini Architecture

When working with code generators, at some point it becomes necessary to choose a target machine. Up to this point we have been reluctant to do so because we wanted the discussion to be as general as possible, so that the concepts could be applied to a variety of architectures. However, in this section

we will work with an example of a code generator, and it now becomes necessary to specify a target machine architecture. It is tempting to choose a popular machine such as a RISC, Intel, Motorola, IBM, or Sparc CPU. If we did so, the student who had access to that processor could conceivably generate executable code for that machine. But what about those who do not have access to the chosen processor? Also, there would be details, such as Object formats (the input to the linker), and supervisor or system calls for certain operations, which we have not explained.

For these reasons, we choose our own simulated machine. This is an architecture which we will specify for the student. We also provide a simulator for this machine, written in the C language. Thus, anyone who has a C compiler has access to our simulated machine, regardless of the actual platform on which it is running. Another advantage of a simulated architecture is that we can make it as simple as necessary to illustrate the concepts of code generation. We need not be concerned with efficiency or completeness. The architecture will be relatively simple and not cluttered with unnecessary features.

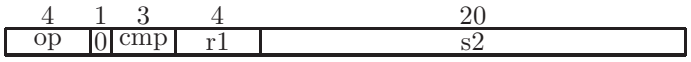
6.5.1 Mini: The Simulated Architecture

In this section we devise a completely fictitious computer, and we provide a simulator for that computer so that the student will be able to generate and execute machine language programs. We call our machine Mini, not because it is supposed to be a 'minicomputer', but because it is really a minimal computer. We have described and implemented just enough of the architecture to enable us to implement a fairly simple code generator. The student should feel free to implement additional features in the Mini architecture. For example, the Mini architecture contains no integer arithmetic; all arithmetic is done with floating-point values, but the instruction set could easily be extended to include integer arithmetic.

The Mini architecture has a 32-bit word size, with 32-bit registers, and a word addressable memory consisting of, at most, 4 G (32 bit) words (the simulator defines a memory of 64 K words, though this is easily extended). There are two addressing modes in the Mini architecture: absolute and register-displacement. In absolute mode, the memory address is stored in the instruction as a 20-bit quantity (in this mode it is only possible to address the lowest megaword of memory). In register-displacement mode, the memory address is computed by adding the contents of the specified general register to the value of the 16-bit offset, or displacement, in the instruction (in this mode it is possible to address all of memory).

The CPU has sixteen general purpose registers and sixteen floating-point registers. All floating-point arithmetic must be done in the floating-point registers (floating-point data are stored in the format of the simulator's host machine, so the student need not be concerned with the specifics of floating-point data formats). There is also a 1-bit flag in the CPU which is set by the compare (CMP) instruction and tested by the conditional branch (JMP) instruction. There is also a 32-bit program counter register (PC). The Mini processor has

Absolute Mode



Register-displacement Mode

Figure 6.11: Mini instruction formats

two instruction formats corresponding to the two addressing modes, as shown in Figure 6.11.

The absolute mode instruction can be described as:
$fpreg[r1] \leftarrow fpreg[r1] opmemory[s2]$
and the register-displacement mode instruction can be described as
$fpreg[r1] \leftarrow fpreg[r1] opmemory[reg[r2] + d2]$.
The operation codes (specified in the op field) are shown below:

0 CLR fpreg[r1] ← 0	Clear Floating-Point Reg.
1 ADD fpreg[r1] ← fpreg[r1] + memory[s2]	Floating-Point Add
2 SUB fpreg[r1] ← fpreg[r1] - memory[s2]	Floating-Point Subtract
3 MUL fpreg[r1] ← fpreg[r1] * memory[s2]	Floating-Point Multiply
4 DIV fpreg[r1] ← fpreg[r1] / memory[s2]	Floating-Point Division
5 JMP PC ← s2 if flag is true	Conditional Branch
6 CMP flag ← r1 cmp memory[s2]	Compare, Set Flag
7 LOD fpreg[r1] ← memory[s2]	Load Floating-Point Register
8 STO memory[s2] ← fpreg[r1]	Store Floating-Point Register
9 HLT	Halt Processor

The Compare field in either instruction format (cmp) is used only by the Compare instruction to indicate the kind of comparison to be done on arithmetic data. In addition to a code of 0, which always sets the flag to True, there are six valid comparison codes as shown below:

1	==	4	<=
2	<	5	>=
3	>	6	!=

The following example of a Mini program will replace the memory word at location 0 with its absolute value. The memory contents are shown in hexadecimal, and program execution is assumed to begin at memory location 1.

```
Loc      Contents
0        00000000              Data       0
1        00100000              CLR        r1              Put 0 into register r1.
2        64100000              CMP        r1,Data,4       Is 0 <= Data?
3        50000006              JMP        Stop            If so, finished.
4        20100000              SUB        r1,Data         If not, find 0-Data.
5        80100000              STO        r1,Data
6        90000000       Stop   HLT                        Halt processor
```

The simulator for the Mini architecture is shown in Appendix C.

6.5.2 The Input to the Code Generator

In our example, the input to the code generator will be a file in which each record is an atom, as discussed in chapters 4 and 5. Here we specify the meaning of the atoms more precisely in the table below:

```
Class Name Operands                          Meaning
1     ADD  left   right  result
                                             result =  left + right
2     SUB  left   right  result
                                             result =  left - right
3     MUL  left   right  result
                                             result =  left * right
4     DIV  left   right  result
                                             result =  left / right
5     JMP  -      -      -       -  dest
                                             branch to dest
10    NEG  left   -      result
                                             result =  - left
11    LBL  -      -      -       -  dest
                                             (no action)
12    TST  left   right  -       cmp dest
                                             branch to dest if
                                             left cmp right is true
13    MOV  left   -      result  -   -
                                             result = left
```

Each atom class is specified with an integer code, and each record may have up to six fields specifying the atom class, the location of the left operand, the location of the right operand, the location of the result, a comparison code (for TST atoms only), and a destination (for JMP, LBL, and TST atoms only). Note that a JMP atom is an unconditional branch, whereas a JMP instruction is a

conditional branch. An example of an input file of atoms which would replace the value of Data with its absolute value is shown below:

```
TST     0     Data     4     L1     -        Branch to L1 if 0 <= Data
NEG     Data  -        Data  -      -        Data = - Data
LBL     L1    -        -     -      -
```

6.5.3 The Code Generator for Mini

The complete code generator is shown in Appendix B, in which the function name is code_gen(). In this section we explain the techniques used and the design of that program. The code generator reads from a file of atoms, and it is designed to work in two passes. Since instructions are 32 bits, the code generator declares integer quantities as long (assuming that the host machine will implement these in 32 bits).

In the first pass it builds a table of Labels, assigning each Label a value corresponding to its ultimate machine address; the table is built by the function build_labels(), and the name of the table is labels. It is simply an array of integers holding the value of each Label. The integer variable pc is used to maintain a hypothetical program counter as the atoms are read, incremented by two for MOV and JMP atoms and incremented by three for all other atoms. The global variable end_data indicates the memory location where the program instructions will begin, since all constants and program variables are stored, beginning at memory location 0, by a function called out_mem() and precede the instructions.

After the first pass is complete, the file of atoms is closed and reopened to begin reading atoms for the second pass. The control structure for the second pass is a switch statement that uses the atom class to determine flow of control. Each atom class is handled by two or three calls to a function that actually generates an instruction - gen(). Label definitions can be ignored in the second pass.

The function which generates instructions takes four arguments:

gen (op, r, add, cmp)

where op is the operation code of the instruction, r is the register for the first operand, add is the absolute address for the second operand, and cmp is the comparison code for Compare instructions. For simplicity, the addressing mode is assumed always to be absolute (this limits us to a one megaword address space). As an example, Figure 6.11 shows that a Multiply atom would be translated by three calls to the gen() function to generate LOD, MUL, and STO instructions.

In Figure 6.11, the function reg() returns an available floating-point register. For simplicity, our implementation of reg() always returns a 1, which means that floating-point values are always kept in floating-point register 1. The structure

inp is used to hold the atom which is being processed. The dest field of an atom is the destination label for jump instructions, and the actual address is obtained from the labels table by a function called lookup(). The code generator sends all instructions to the standard output file as hex characters, so that the user has the option of discarding them, storing them in a file, or piping them directly into the Mini simulator. The generated instructions are shown to the right in Figure 6.11.

The student is encouraged to use, abuse, modify and/or distribute (but not for profit) the software shown in the Appendix to gain a better understanding of the operation of the code generator.

6.5.4 Exercises

1. How is the compiler's task simplified by the fact that floating-point is the only numeric data type in the Mini architecture?

2. Disassemble the following Mini instructions. Assume that general register 7 contains hex 20, and that the variables A and B are stored at locations hex 21 and hex 22, respectively.

```
70100021
10300022
18370002
```

3. Show the code, in hex, generated by the code generator for each of the following atom strings. Assume that A and B are stored at locations 0 and 1, respectively. Allocate space for the temporary value T1 at the end of the program.

   (a)

class	left	right	result	cmp	dest
MUL	A	B	T1	-	-
LBL	-	-	-	-	L1
TST	A	T1	-	2	L1
JMP	-	-	-	-	L2
MOVT1	-	B	-	-	
LBL	-	-	-	-	L2

   (b)

class	left	right	result	cmp	dest
NEG	A	-	T1	-	-
LBL	-	-	-	0	L1
MOVT1	-	B	-	-	
TST	B	T1	-	4	L1

(c)

class	left	right	result	cmp	dest
TST	A	B	-	6	L2
JMP	-	-	-	-	L1
LBL	-	-	-	-	L2
TST	A	T1	-	0	L2
LBL	-	-	-	-	L1

6.6 Chapter Summary

This chapter commences our study of the back end of a compiler. Prior to this point everything we have studied was included in the front end. The code generator is the portion of the compiler which accepts syntax trees or atoms (sometimes referred to as 3 -address code) created by the front end and converts them to machine language instructions for the target machine.

It was shown that if the language of syntax trees or atoms (known as an intermediate form) is standardized, then, as new machines are constructed, we need only rewrite the back ends of our compilers. Conversely, as new languages are developed, we need only rewrite the front ends of our compilers.

The process of converting atoms to instructions is relatively easy to implement, since each atom corresponds to a small, fixed number of instructions. The main problems to be solved in this process are (1)obtaining memory addresses for forward references and (2) register allocation . Forward references result from branch instructions to a higher memory address which can be computed by either single pass or multiple pass methods. With a single pass method, a fixup table for forward references is required. For either method a table of labels is used to bind labels to target machine addresses.

Register allocation is important for efficient object code in machines which have several CPU registers. An algorithm for allocating registers from syntax trees are presented. Algorithms which make use of common subexpressions in an expression, or common subexpressions in a block of code, will be discussed in Chapter 7 .

This chapter concludes with a case study code generator. This code generator can be used for any compiler whose front end puts out atoms as we have defined them. In order to complete the case study, we define a fictitious target machine, called Mini. This machine has a very simple 32 bit architecture, which simplifies the code generation task. Since we have a simulator for the Mini machine, written in the C language, in Appendix C, anyone with access to a C compiler can run the Mini machine.

It is assumed that all arithmetic is done in floating-point format, which eliminates the need for data conversions. Code is generated by a function with

three arguments specifying the operation code and two operands. The code generator, shown in Appendix B.3, uses a two pass method to handle forward references.

Chapter 7

Optimization

7.1 Introduction and View of Optimization

In recent years, most research and development in the area of compiler design has been focused on the optimization phases of the compiler. Optimization is the process of improving generated code so as to reduce its potential running time and/or reduce the space required to store it in memory. Software designers are often faced with decisions which involve a space-time tradeoff, i.e. one method will result in a faster program, another method will result in a program which requires less memory, but no method will do both. However, many optimization techniques are capable of improving the object program in both time and space, which is why they are employed in most modern compilers. This results from either the fact that much effort has been directed toward the development of optimization techniques, or from the fact that the code normally generated is very poor and easily improved.

The word *optimization* is possibly a misnomer, since the techniques that have been developed simply attempt to improve the generated code, and few of them are guaranteed to produce, in any sense, optimal (the most efficient possible) code. Nevertheless, the word optimization is the one that is universally used to describe these techniques, and we will use it also. We have already seen that some of these techniques (such as register allocation) are normally handled in the code generation phase, and we will not discuss them here.

Optimization techniques can be separated into two general classes: local and global. Local optimization techniques normally are concerned with transformations on small sections of code (involving only a few instructions) and generally operate on the machine language instructions which are produced by the code generator. On the other hand, global optimization techniques are generally concerned with larger blocks of code, or even multiple blocks or modules, and will be applied to the intermediate form, atom strings, or syntax trees put out by the parser. Both local and global optimization phases are optional, but may be included in the compiler as shown in Figure 7.1, i.e., the output of the parser is

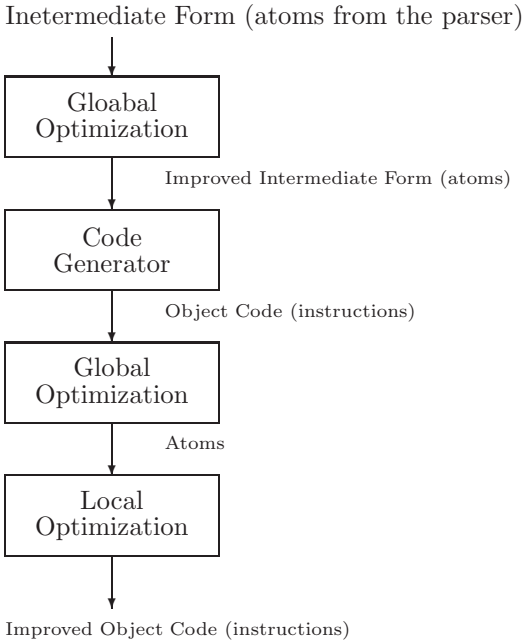


Figure 7.1: Sequence of Optimization Phases in a Compiler

the input to the global optimization phase, the output of the global optimization phase is the input to the code generator, the output of the code generator is the input to the local optimization phase, and the output of the local optimization phase is the final output of the compiler. The three compiler phases shown in Figure 7.1 make up the back end of the compiler, discussed in Section 6.1.

In this discussion on improving performance, we stress the single most important property of a compiler - that it preserve the semantics of the source program. In other words, the purpose and behavior of the object program should be exactly as specified by the source program for all possible inputs. There are no conceivable improvements in efficiency which can justify violating this promise.

Having made this point, there are frequently situations in which the computation specified by the source program is ambiguous or unclear for a particular computer architecture. For example, in the expression $(a + b) * (c + d)$ the compiler will have to decide which addition is to be performed first (assuming that the target machine has only one Arithmetic and Logic Unit). Most programming languages leave this unspecified, and it is entirely up to the compiler

designer, so that different compilers could evaluate this expression in different ways. In most cases it may not matter, but if any of a, b, c, or d happen to be function calls which produce output or side effects, it may make a significant difference. Languages such as Java, C, Lisp, and APL, which have assignment operators, yield an even more interesting example:

$a = 2; b = (a * 1 + (a = 3))$;

Some compiler writers feel that programmers who use ambiguous expressions such as these deserve whatever the compiler may do to them.

A fundamental question of philosophy is inevitable in the design of the optimization phases. Should the compiler make extensive transformations and improvements to the source program, or should it respect the programmer's decision to do things that are inefficient or unnecessary? Most compilers tend to assume that the average programmer does not intentionally write inefficient code, and will perform the optimizing transformations. A sophisticated programmer or hacker who, in rare cases, has a reason for writing the code in that fashion can usually find a way to force the compiler to generate the desired output.

One significant problem for the user of the compiler, introduced by the optimization phases, has to do with debugging. Many of the optimization techniques will remove unnecessary code and move code within the object program to an extent that run-time debugging is affected. The programmer may attempt to step through a series of statements which either do not exist, or occur in an order different from what was originally specified by the source program!

To solve this problem, most modern and available compilers include a switch with which optimization may be turned on or off. When debugging new software, the switch is off, and when the software is fully tested, the switch can be turned on to produce an efficient version of the program for distribution. It is essential, however, that the optimized version and the non-optimized version be functionally equivalent (i.e., given the same inputs, they should produce identical outputs). This is one of the more difficult problems that the compiler designer must deal with.

Another solution to this problem, used by IBM in the early 1970's for its PL/1 compiler, is to produce two separate compilers. The checkout compiler was designed for interactive use and debugging. The optimizing compiler contained extensive optimization, but was not amenable to the testing and development of software. Again, the vendor (IBM in this case) had to be certain that the two compilers produced functionally equivalent output.

7.1.1 Exercises

1. Using a Java compiler,

   (a) what would be printed as a result of running the following:

```
{
   int a, b;
```

```
        b = (a = 2) + (a = 3);
        System.out.println ("a is " + a);
    }
```

(b) What other value might be printed as a result of compilation with a different compiler?

2. Explain why the following two statements cannot be assumed to be equivalent:

```
a = f(x) + f(x) + f(x) ;
```

```
a = 3 * f(x) ;
```

3. (a) Perform the following computations, rounding to four significant digits after each operation.

```
(0.7043 + 0.4045) + -0.3330 = ?
```

```
0.7043 + (0.4045 +  -0.3330) = ?
```

(b) What can you can conclude about the associativity of addition with computer arithmetic?

7.2 Global Optimization

As mentioned previously, *global optimization* is a transformation on the output of the parser. Global optimization techniques will normally accept, as input, the intermediate form as a sequence of atoms (three-address code) or syntax trees. There are several global optimization techniques in the literature - more than we can hope to cover in detail. Therefore, we will look at the optimization of common subexpressions in basic blocks in some detail, and then briefly survey some of the other global optimization techniques.

A few optimization techniques, such as algebraic optimizations, can be considered either local or global. Since it is generally easier to deal with atoms than with instructions, we will include algebraic techniques in this section.

7.2.1 Basic Blocks and DAGs

The sequence of atoms put out by the parser is clearly not an optimal sequence; there are many unnecessary and redundant atoms. For example, consider the Java statement:

```
a = (b + c) * (b + c) ;
```

```
(ADD, b, c, T1)
(ADD, b, c, T2)
(MUL, T1, T2, T3)
(MOV, T3,, a)
```

Figure 7.2: Atom Sequence for $a = (b + c) * (b + c)$;

```
(ADD, b, c, T1)
(MUL, T1, T1, a)
```

Figure 7.3: Optimized Atom Sequence for $a = (b + c) * (b + c)$;

The sequence of atoms put out by the parser could conceivably be as shown in Figure 7.2.

Every time the parser finds a correctly formed addition operation with two operands it blindly puts out an ADD atom, whether or not this is necessary. In the above example, it is clearly not necessary to evaluate the sum b + c twice. In addition, the MOV atom is not necessary because the MUL atom could store its result directly into the variable a. The atom sequence shown in Figure 7.3 is equivalent to the one given in Figure 7.2, but requires only two atoms because it makes use of common subexpressions and it stores the result in the variable a, rather than a temporary location.

In this section, we will demonstrate some techniques for implementing these optimization improvements to the atoms put out by the parser. These improvements will result in programs which are both smaller and faster, i.e., they optimize in both space and time.

It is important to recognize that these optimizations would not have been possible if there had been intervening Label or Jump atoms in the parser output. For example, if the atom sequence had been as shown in Figure 7.4, we could not have optimized to the sequence of Figure 7.3, because there could be atoms which jump into this code at Label L1, thus altering our assumptions about the values of the variables and temporary locations. (The atoms in Figure 7.4 do not result from the given Java statement, and the example is, admittedly, artificially contrived to make the point that Label atoms will affect our ability to optimize.)

By the same reasoning, Jump or Branch atoms will interfere with our ability to make these optimizing transformations to the atom sequence. In Figure 7.4 the MUL atom cannot store its result into the variable a, because the compiler does not know whether the conditional branch will be taken.

The optimization techniques which we will demonstrate can be effected only in certain subsequences of the atom string, which we call *basic blocks*. A basic block is a section of atoms which contains no Label or branch atoms (i.e., LBL, TST, JMP). In Figure 7.5, we show that the atom sequence of Figure 7.4 is divided into three basic blocks.

```
(ADD, b, c, T1)
(LBL, L1)
(ADD, b, c, T2)
(MUL, T1, T2, T3)
(TST, b, c,, 1, L3)
(MOV, T3,, a)
```

Figure 7.4: Example of an Atom Sequence Which Cannot be Optimized

(ADD, b, c, T1)	Block 1
(LBL, L1)	
(ADD, b, c, T2	Block 2
(MUL, T1, T2, T3	
(TST, b, c,, 1, L3)	
(MOV, T3,, a)	Block 3

Figure 7.5: Basic blocks contain No LBL, TST, or JMP atoms

Each basic block is optimized as a separate entity. There are more advanced techniques which permit optimization across basic blocks, but they are beyond the scope of this text. We use a *Directed Acyclic Graph*, or *DAG*, to implement this optimization. The DAG is directed because the arcs have arrows indicating the direction of the arcs, and it is acyclic because there is no path leading from a node back to itself (i.e., it has no cycles). The DAG is similar to a syntax tree, but it is not truly a tree because some nodes may have more than one parent and also because the children of a node need not be distinct. An example of a DAG, in which interior nodes are labeled with operations, and leaf nodes are labeled with operands is shown in Figure 7.6.

Each of the operations in Figure 7.6 is a binary operation (i.e., each operation has two operands), consequently each interior node has two arcs pointing to the two operands. Note that in general we will distinguish between the left and right arc because we need to distinguish between the left and right operands of an operation (this is certainly true for subtraction and division, which are not commutative operations). We will be careful to draw the DAGs so that it is always clear which arc represents the left operand and which arc represents the right operand. For example, in Figure 7.6 the left operand of the addition labeled T3 is T2, and the right operand is T1. Our plan is to show how to build a DAG from an atom sequence, from which we can then optimize the atom sequence.

We will begin by building DAGs for simple arithmetic expressions. DAGs can also be used to optimize complete assignment statements and blocks of statements, but we will not take the time to do that here. To build a DAG, given a sequence of atoms representing an arithmetic expression with binary operations, we use the following algorithm:

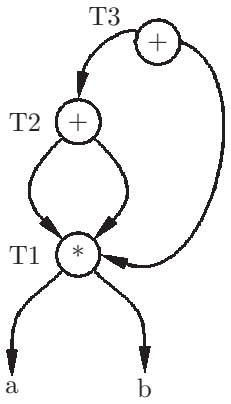


Figure 7.6: An example of a DAG

1. Read an atom.
2. If the operation and operands match part of the existing DAG (i.e., if they form a sub DAG), then add the result Label to the list of Labels on the parent and repeat from Step 1. Otherwise, allocate a new node for each operand that is not already in the DAG, and a node for the operation. Label the operation node with the name of the result of the operation.
3. Connect the operation node to the two operands with directed arcs, so that it is clear which operand is the left and which is the right.
4. Repeat from Step 1.

As an example, we will build a DAG for the expression `a * b + a * b + a * b`. This expression clearly has some common subexpressions, which should make it amenable for optimization. The atom sequence as put out by the parser would be:

```
(MUL, a, b, T1)
(MUL, a, b, T2)
(ADD, T1, T2, T3)
(MUL, a, b, T4)
(ADD, T3, T4, T5)
```

We follow the algorithm to build the DAG, as shown in Figure 7.7, in which we show how the DAG is constructed as each atom is processed.

The DAG is a graphical representation of the computation needed to evaluate the original expression in which we have identified common subexpressions. For example, the expression `a * b` occurs three times in the original expression `a * b + a * b + a * b`. The three atoms corresponding to these subexpressions store results into T1, T2, and T4. Since the computation need be done only once, these three atoms are combined into one node in the DAG labeled T1.2.4.

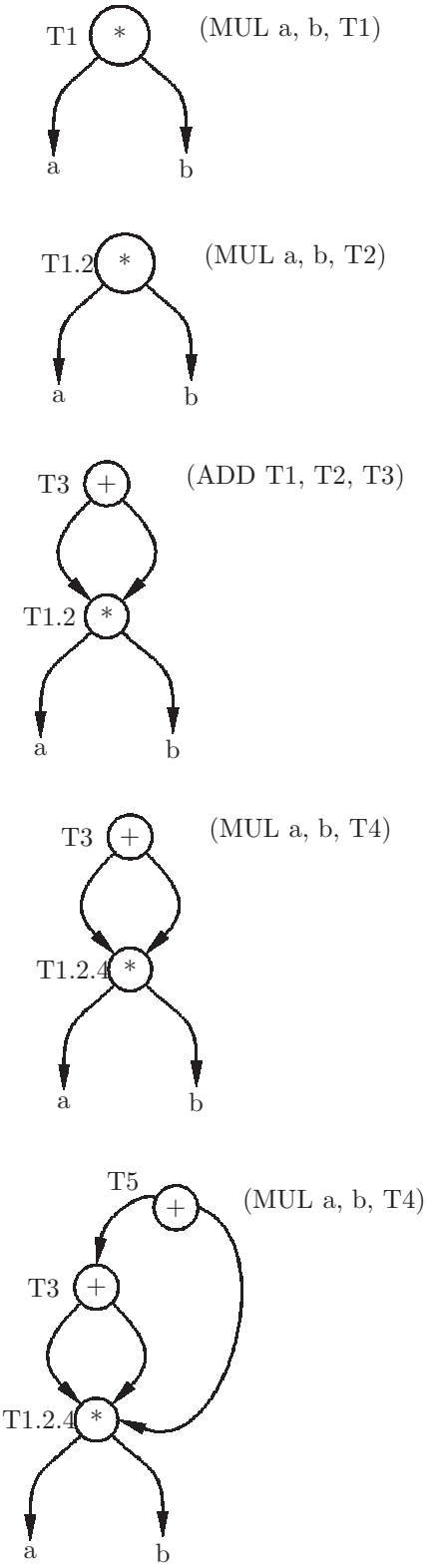


Figure 7.7: Building the DAG for `a * b + a * b + a * b`

After that point, any atom which uses T1, T2, or T4 as an operand will point to T1.2.4.

We are now ready to convert the DAG to a basic block of atoms. The algorithm given below will generate atoms (in reverse order) in which all common subexpressions are evaluated only once:

1. Choose any node having no incoming arcs (initially there should be only one such node, representing the value of the entire expression).
2. Put out an atom for its operation and its operands.
3. Delete this node and its outgoing arcs from the DAG.
4. Repeat from Step 1 as long as there are still operation nodes remaining in the DAG.

This algorithm is demonstrated in Figure 7.8. in which we are working with the same expression that generated the DAG of Figure 7.7. The DAG and the output are shown for each iteration of the algorithm (there are three iterations).

A composite node, such as T1.2.4, is referred to by its full name rather than simply T1 or T2 by convention, and to help check for mistakes. The student should verify that the three atoms generated in Figure 7.8 actually compute the given expression, reading the atoms from bottom to top. We started with a string of five atoms, and have improved it to an equivalent string of only three atoms. This will result in significant savings in both run time and space required for the object program.

Unary operations can be handled easily using this method. Since a unary operation has only one operand, its node will have only one arc pointing to the operand, but in all other respects the algorithms given for building DAGs and generating optimized atom sequences remain unchanged. Consequently, this method generalizes well to expressions involving operations with any number of operands, though for our purposes operations will generally have two operands.

Sample Problem 7.2.1

Construct the DAG and show the optimized sequence of atoms for the Java expression `(a - b) * c + d * (a - b) * c`. *The atoms produced by the parser are shown below:*

```
(SUB, a, b, T1)
(MUL, T1, c, T2)
(SUB, a, b, T3)
(MUL, d, T3, T4)
(MUL, T4, c, T5)
(ADD, T2, T5, T6)
```

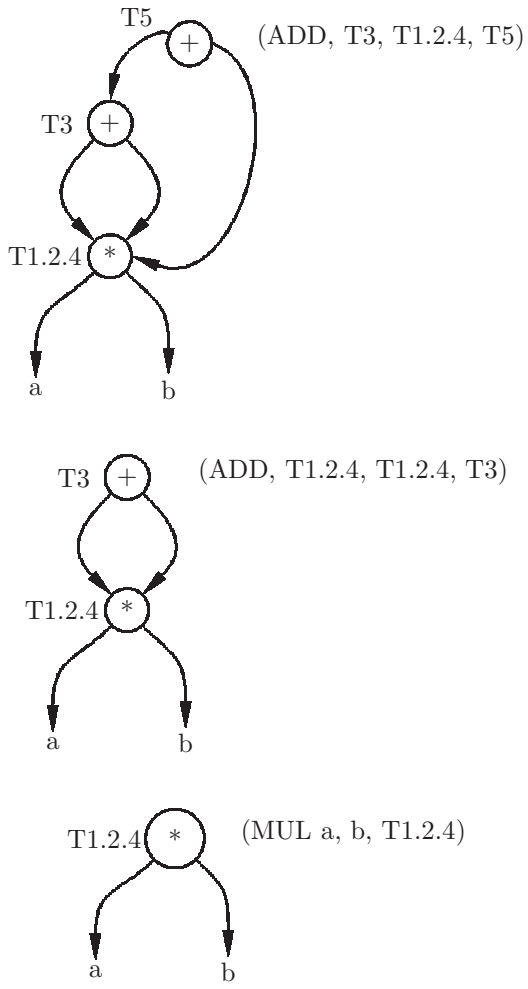
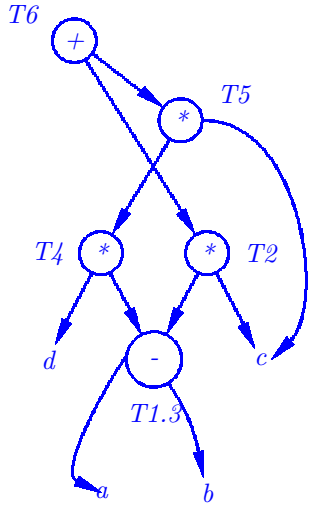


Figure 7.8: Generating atoms from the DAG for `a * b + a * b + a * b`

Solution:

```
   (SUB, a, b, T1.3)
(MUL, d, T1.3, T4)
(MUL, T4, c, T5)
(MUL, T1.3, c, T2)
(ADD, T2, T5, T6)
```



7.2.2 Other Global Optimization Techniques

We will now examine a few other common global optimization techniques; however, we will not go into the implementation of these techniques.

Unreachable code is an atom or sequence of atoms which cannot be executed because there is no way for the flow of control to reach that sequence of atoms. For example, in the following atom sequence the MUL, SUB, and ADD atoms will never be executed because of the unconditional jump preceding them:

```
(JMP, L1)
(MUL, a, b, T1)
(SUB, T1, c, T2)
(ADD, T2, d, T3)
(LBL, L2)
```

Thus, the three atoms following the JMP and preceding the LBL can all be removed from the program without changing the purpose of the program:

```
{
    a = b + c * d; // This statement has no effect and can be removed.
    b = c * d / 3;
    c = b - 3;
    a = b - c;
    System.out.println (a + b + c);
}
```

Figure 7.9: Elimination of Dead Code

```
(JMP, L1)
(LBL, L2)
```

In general, a JMP atom should always be followed by a LBL atom. If this is not the case, simply remove the intervening atoms between the JMP and the next LBL.

Data flow analysis is a formal way of tracing the way information about data items moves through the program and is used for many optimization techniques. Though data flow analysis is beyond the scope of this text, we will look at some of the optimizations that can result from this kind of analysis.

One such optimization technique is *elimination of dead code*, which involves determining whether computations specified in the source program are actually used and affect the program's output. For example, the program in Figure 7.9 contains an assigment to the variable `a` which has no effect on the output since `a` is not used subsequently, but prior to another assignment to the variable `a`.

Another optimization technique which makes use of data flow analysis is the detection of loop invariants. A *loop invariant* is code within a loop which deals with data values that remain constant as the loop repeats. Such code can be moved outside the loop, causing improved run time without changing the program's semantics. An example of loop invariant code is the call to the square root function (sqrt) in the program of Figure 7.10.

Since the value assigned to `a` is the same each time the loop repeats, there is no need for it to be repeated; it can be done once before entering the loop (we need to be sure, however, that the loop is certain to be executed at least once). This optimization will eliminate 999 unnecessary calls to the sqrt function.

The remaining global optimization techniques to be examined in this section all involve mathematical transformations. The student is cautioned that their use is not universally recommended, and that it is often possible, by employing them, that the compiler designer is effecting transformations which are undesirable to the source programmer. For example, the question of the meaning of arithmetic overflow is crucial here. If the unoptimized program reaches an overflow condition for a particular input, is it valid for the optimized program to avoid the overflow? (Be careful; most computers have run-time traps designed to transfer control to handle conditions such as overflow. It could be

```
{
   for (int i=0; i<1000; i++)
      { a = sqrt (x);          // loop invariant
        vector[i] = i * a;
      }
}

{
   a = sqrt (x);              // loop invariant
   for (int i=0; i<1000; i++)
      {
        vector[i] = i * a;
      }
}
```

Figure 7.10: Movement of Loop Invariant Code

that the programmer intended to trap certain input conditions.) There is no right or wrong answer to this question, but it is an important consideration when implementing optimization.

Constant folding is the process of detecting operations on constants, which could be done at compile time rather than run time. An example is shown in Figure 7.11 in which the value of the variable `a` is known to be 6, and the value of the expression `a * a` is known to be 36. If these computations occur in a small loop, constant folding can result in significant improvement in run time (at the expense of a little compile time).

Another mathematical transformation is called *reduction in strength*. This optimization results from the fact that certain operations require more time

```
{
   a = 2 * 3;           // a must be 6
   b = c + a * a;       // a*a must be 36
}


{
   a = 6;               // a must be 6
   b = c + 36;          // a*a must be 36
}
```

Figure 7.11: Constant Folding

`a + b == b + a`	Addition is commutative
`(a + b) + c == a + (b + c)`	Addition is Associative
`a * (b + c) == a * b + a * c`	Multiplication distributes over addition

Figure 7.12: Algebraic Identities

than others on virtually all architectures. For example, multiplication can be expected to be significantly more time consuming than addition. Thus, the multiplication `2 * x` is likely to be slower than the addition `x + x`. Likewise, if there is an exponentiation operator, `x**2` is certain to be slower than `x * x`.

A similar use of reduction in strength involves using the shift instructions available on most architectures to speed up fixed point multiplication and division. A multiplication by a positive power of two is equivalent to a left shift, and a division by a positive power of two is equivalent to a right shift. For example, the multiplication `x*8` can be done faster simply by shifting the value of `x` three bit positions to the left, and the division `x/32` can be done faster by shifting the value of `x` five bit positions to the right.

Our final example of mathematical transformations involves *algebraic transformations* using properties such as commutativity, associativity, and the distributive property, all summarized in Figure 7.12.

We do not believe that these properties are necessarily true when dealing with computer arithmetic, due to the finite precision of numeric data. Nevertheless, they are employed in many compilers, so we give a brief discussion of them here.

Though these properties are certainly true in mathematics, they do not necessarily hold in computer arithmetic, which has finite precision and is subject to overflow in both fixed-point and floating-point representations. Thus, the decision to make use of these properties must take into consideration the programs which will behave differently with optimization put into effect. At the very least, a warning to the user is recommended for the compiler's user manual.

The discussion of common subexpresssions in Section 7.2.1 would not have recognized any common subexpressions in the following:

```
a = b + c;
b = c + d + b;
```

but by employing the commutative property, we can eliminate an unnecessary computation of `b + c`

```
a = b + c;
b = a + d;
```

A multiplication operation can be eliminated from the expression `a * c + b * c` by using the distributive property to obtain `(a + b) * c`.

Compiler writers who employ these techniques create more efficient programs for the large number of programmers who want and appreciate the improvements, but risk generating unwanted code for the small number of programmers

who require that algebraic expressions be evaluated exactly as specified in the source program.

Sample Problem 7.2.2

Use the methods of unreachable code, constant folding, reduction in strength, loop invariants, and dead code to optimize the following atom stream; you may assume that the TST condition is initially not satisfied:

```
(LBL, L1)
(TST, a, b,, 1, L2)
(SUB, a, 1, a)
(MUL, x, 2, b)
(ADD, x, y, z)
(ADD, 2, 3, z)
(JMP, L1)
(SUB, a, b, a)
(MUL, x, 2, z)
(LBL, L2)
```

Solution:

```
(LBL, L1)
(TST, a, b,, 1, L2)
(SUB, a, 1, a)
(MUL, x, 2, b)              Reduction in strength
(ADD, x, y, z)              Elimination of dead code
(ADD, 2, 3, z)              Constant folding, loop invariant
(JMP, L1)
(SUB, a, b, a)              Unreachable code
(MUL, x, 2, z)              Unreachable code
(LBL, L2)
```

```
(MOV, 5,, z)
(LBL, L1)
(TST, a, b,, 1, L2)
(SUB, a, 1, a)
```

```
(ADD, x, x, b)
(JMP, L1)
(LBL, L2)
```

7.2.3 Exercises

1. Eliminate *common subexpressions* from each of the following strings of atoms, using DAGs as shown in Sample Problem **??** (we also give the Java expressions from which the atom strings were generated):

   (a) `(b + c) * d * (b + c)`

   ```
   (ADD, b, c, T1)
   (MUL, T1, d, T2)
   (ADD, b, c, T3)
   (MUL, T2, T3, T4)
   ```

   (b) `(a + b) * c / ((a + b) * c - d)`

   ```
   (ADD, a, b, T1)
   (MUL, T1, c, T2)
   (ADD, a, b, T3)
   (MUL, T3, c, T4)
   (SUB, T4, d, T5)
   (DIV, T2, T5, T6)
   ```

   (c) `(a + b) * (a + b) - (a + b) * (a + b)`

   ```
   (ADD, a, b, T1)
   (ADD, a, b, T2)
   (MUL, T1, T2, T3)
   (ADD, a, b, T4)
   (ADD, a, b, T5)
   (MUL, T4, T5, T6)
   (SUB, T3, T6, T7)
   ```

   (d) `((a + b) + c) / (a + b + c) - (a + b + c)`

```
(ADD, a, b, T1)
(ADD, T1, c, T2)
(ADD, a, b, T3)
(ADD, T3, c, T4)
(DIV, T2, T4, T5)
(ADD, a, b, T6)
(ADD, T6, c, T7)
(SUB, T5, T7, T8)
```

(e) `a / b - c / d - e / f`

```
(DIV, a, b, T1)
(DIV, c, d, T2)
(SUB, T1, T2, T3)
(DIV, e, f, T4)
(SUB, T3, T4, T5)
```

2. How many different atom sequences can be generated from the DAG given in your response to Problem 1 (e), above?

3. In each of the following sequences of atoms, eliminate the *unreachable* atoms: (a)

```
(ADD, a, b, T1)
(LBL, L1)
(SUB, b, a, b)
(TST, a, b,, 1, L1)
(ADD, a, b, T3)
(JMP, L1)
```

(b)

```
(ADD, a, b, T1)
(LBL, L1)
(SUB, b, a, b)
(JMP, L1)
(ADD, a, b, T3)
(LBL, L2)
```

(c)

```
(JMP, L2)
(ADD, a, b, T1)
(TST, a, b,, 3, L2)
```

```
(SUB, b, b, T3)
(LBL, L2)
(MUL, a, b, T4)
```

4. In each of the following Java methods, eliminate statements which constitute *dead code*.

   (a)

```
int f (int  d)
{ int a,b,c;
     a = 3;
     b = 4;
     d = a * b + d;
     return d;
}
```

   (b)

```
int f (int d)
{ int a,b,c;
     a = 3;
     b = 4;
     c = a +b;
     d = a + b;
     a = b + c * d;
     b = a + c;
     return d;
}
```

5. In each of the following Java program segments, optimize the loops by moving *loop invariant code* outside the loop:

   (a)

```
{     for (i=0; i<100; i++)
         {     a = x[i] + 2 * a;
               b = x[i];
               c = sqrt (100 * c);
         }
}
```

   (b)

```
{     for (j=0; j<50; j++)
        {     a = sqrt (x);
          n = n * 2;
```

```
            for (i=0; i<10; i++)
                {     y = x;
                      b[n] = 0;
                      b[i] = 0;
                }
            }
}
```

6. Show how *constant folding* can be used to optimize the following Java program segments:

   (a)

```
a = 2 + 3 * 8;
b = b + (a - 3);
```

   (b)

```
int f (int  c)
{     final int a = 44;
     final int b = a - 12;
     c = a + b - 7;
     return c;
}
```

7. Use *reduction in strength* to optimize the following sequences of atoms. Assume that there are `(SHL, x, y, z)` and `(SHR, x, y, z)` atoms which will shift x left or right respectively by y bit positions, leaving the result in z (also assume that these are fixed-point operations):

   (a)

```
(MUL, x, 2, T1)
(MUL, y, 2, T2)
```

   (b)

```
(MUL, x, 8, T1)
(DIV, y, 16, T2)
```

8. Which of the following optimization techniques, when applied successfully, will always result in improved execution time? Which will result in reduced program size?

   (a) Detection of common subexpressions with DAGs
   (b) Elimination of unreachable code
   (c) Elimination of dead code
   (d) Movement of loop invariants outside of loop
   (e) Constant folding
   (f) Reduction in strength

7.3 Local Optimization

In this section we discuss local optimization techniques. The definition of local versus global techniques varies considerably among compiler design textbooks. Our view is that any optimization which is applied to the generated code is considered local. Local optimization techniques are often called peephole optimization, since they generally involve transformations on instructions which are close together in the object program. The student can visualize them as if peering through a small peephole at the generated code.

There are three types of local optimization techniques which will be discussed here: load/store optimization, jump over jump optimization, and simple algebraic optimization. In addition, register allocation schemes such as the one discussed in Section 6.4 could be considered local optimization, though they are generally handled in the code generator itself.

The parser would translate the expression a + b - c into the following stream of atoms:

```
(ADD, a, b, T1)
(SUB, T1, c, T2)
```

The simplest code generator design, as presented in Chapter 6, would generate three instructions corresponding to each atom:

1. Load the first operand into a register (LOD)
2. Perform the operation
3. Store the result back to memory (STO).

The code generator would then produce the following instructions from the atoms:

```
LOD     R1,a
ADD     R1,b
STO     R1,T1
LOD     R1,T1
SUB     R1,c
STO     R1,T2
```

Notice that the third and fourth instructions in this sequence are entirely unnecessary since the value being stored and loaded is already at its destination. The above sequence of six instructions can be optimized to the following sequence of four instructions by eliminating the intermediate Load and Store instructions as shown below:

```
LOD     R1,a
ADD     R1,b
SUB     R1,c
STO     R1,T2
```

For lack of a better term, we call this a *load/store optimization*. It is clearly machine dependent.

Another local optimization technique, which we call a jump over jump optimization, is very common and has to do with unnecessary jumps. The student has already seen examples in Chapter 4 of conditional jumps in which it is clear that greater efficiency can be obtained by rewriting the conditional logic. A good example of this can be found in a Java compiler for the statement if (a¿b) a = b;. It might be translated into the following stream of atoms:

```
(TST, a, b,, 3, L1)
(JMP, L2)
(LBL, L1)
(MOV, b,, a)
(LBL, L2)
```

A reading of this atom stream is "Test for a greater than b, and if true, jump to the assignment. Otherwise, jump around the assignment." The reason for this somewhat convoluted logic is that the TST atom uses the same comparison code found in the expression. The instructions generated by the code generator from this atom stream would be:

```
     LOD     R1,a
     CMP     R1,b,3                  //Is R1 > b?
     JMP     L1
     CMP     0,0,0                   // Unconditional Jump
     JMP     L2
L1:
     LOD     R1,b
     STO     R1,a
L2:
```

It is not necessary to implement this logic with two Jump instructions. We can improve this code significantly by testing for the condition to be false rather than true, as shown below:

```
     LOD     R1,a
     CMP     R1,b,4                    // Is R1 <= b?
     JMP     L1
     LOD     R1,b
     STO     R1,a
L1:
```

This optimization could have occurred in the intermediate form (i.e., we could have considered it a global optimization), but this kind of jump over jump can occur for various other reasons. For example, in some architectures, a conditional jump is a *short* jump (to a restricted range of addresses), and an unconditional jump is a *long* jump. Thus, it is not known until code has been generated whether the target of a conditional jump is within reach, or whether an unconditional jump is needed to jump that far.

The final example of local optimization techniques involves simple algebraic transformations which are machine dependent and are called simple algebraic optimizations. For example, the following instructions can be eliminated:

```
MUL     R1, 1
ADD     R1, 0
```

because multiplying a value by 1, or adding 0 to a value, should not change that value. (Be sure, though, that the instruction has not been inserted to alter the condition code or flags register.) In addition, the instruction (MUL R1, 0) can be improved by replacing it with (CLR R1), because the result will always be 0 (this is actually a reduction in strength transformation).

7.3.1 Exercises

1. Optimize each of the following code segments for unnecessary Load/Store instructions:

```
(a)                              (b)
   LOD     R1,a                       LOD     R1,a
   ADD     R1,b                       LOD     R2,c
   STO     R1,T1                      ADD     R1,b
   LOD     R1,T1                      ADD     R2,b
   SUB     R1,c                       STO     R2,T1
   STO     R1,T2                      ADD     R1,c
   LOD     R1,T2                      LOD     R2,T1
   STO     R1,d                       STO     R1,T2
                                      STO     R2,c
```

2. Optimize each of the following code segments for unnecessary jump over jump instructions:

(a)

```
          CMP     R1,a,1
          JMP     L1
          CMP     0,0,0
          JMP     L2
     L1:
          ADD     R1,R2
     L2:
```

(b)

```
          CMP     R1,a,5
          JMP     L1
          CMP     0,0,0
          JMP     L2
     L1:
          SUB     R1,a
     L2:
```

(c)

```
     L1:
          ADD     R1,R2
          CMP     R1,R2,3
          JMP     L2
          CMP     0,0,0
          JMP     L1
     L2:
```

3. Use any of the local optimization methods of this section to optimize the following code segment:

```
     CMP     R1,R2,6              // JMP if R1 != R2
     JMP     L1
     CMP     0,0,0
     JMP     L2
L1:
     LOD     R2,a
     ADD     R2,b
     STO     R2,T1
     LOD     R2,T1
     MUL     R2,c
     STO     R2,T2
     LOD     R2,T2
     STO     R2,d
     SUB     R1,0
     STO     R1,b
L2:
```

7.4 Chapter Summary

Optimization has to do with the improvement of machine code and/or intermediate code generated by other phases of the compiler. These improvements can result in reduced run time and/or space for the object program. There are two main classifications of optimization: global and local. Global optimization operates on atoms or syntax trees put out by the front end of the compiler, and local optimization operates on instructions put out by the code generator. The term *optimization* is used for this phase of the compiler, even though it is never certain to produce optimal code in either space or time.

The compiler writer must be careful not to change the intent of the program when applying optimizing techniques. Many of these techniques can have a profound effect on debugging tools; consequently, debugging is generally done on unoptimized code.

Global optimization is applied to blocks of code in the intermediate form (atoms) which contain no Label or branch atoms. These are called basic blocks, and they can be represented by directed acyclic graphs (DAGs), in which each interior node represents an operation with links to its operands. We show how the DAGs can be used to optimize common subexpressions in an arithmetic expression.

We briefly describe a few more global optimization techniques without going into the details of their implementation. They include: (1) unreachable code - code which can never be executed and can therefore be eliminated; (2) dead code - code which may be executed but can not have any effect on the program's output and can therefore be eliminated; (3) loop invariant code - code which is inside a loop, but which doesn't really need to be in the loop and can be moved out of the loop; (4) constant folding: detecting arithmetic operations on constants which can be computed at compile time rather than at run time; (5) reduction in strength: substituting a faster arithmetic operation for a slow one; (6) algebraic transformations: transformations involving the commutative, associative, and distributive properties of arithmetic.

We describe three types of local optimization: (1) load/store optimization - eliminating unnecessary Load and Store instructions in a Load/Store architecture; (2) jump over jump optimizations - replacing two Jump instructions with a single Jump by inverting the logic; (3) simple algebraic optimization - eliminating an addition or subtraction of 0 or a multiplication or division by 1.

These optimization techniques are optional, but they are used in most modern compilers because of the resultant improvements to the object program, which are significant.

Glossary

absolute addressing - An address mode in the Mini architecture which stores a complete memory address in a single instruction field.

action - An executable statement or procedure, often used in association with an automaton or program specification tool.

action symbols - Symbols in a translation grammar enclosed in {braces} and used to indicate output or a procedure call during the parse.

action table - A table in LR parsing algorithms which is used to determine whether a shift or reduce operation is to be performed.

algebraic transformations - An optimization technique which makes use of algebraic properties, such as commutativity and associativity to simplify arithmetic expressions.

alphabet - A set of characters used to make up the strings in a given language.

ambiguous grammar - A grammar which permits more than one derivation tree for a particular input string.

architecture - The definition of a computer's central processing unit as seen by a machine language programmer, including specifications of instruction set operations, instruction formats, addressing modes, data formats, CPU registers, and input/output instruction interrupts and traps.

arithmetic expressions - Infix expressions involving numeric constants, variables, arithmetic operations, and parentheses.

atom - A record put out by the syntax analysis phase of a compiler which specifies a primitive operation and operands.

attributed grammar - A grammar in which each symbol may have zero or more attributes, denoted with subscripts, and each rule may have zero or more attribute computation rules associated with it.

automata theory - The branch of computer science having to do with theoretical machines.

back end - The last few phases of the compiler, code generation and optimization, which are machine dependent.

balanced binary search tree - A binary search tree in which the difference in the heights of both subtrees of each node does not exceed a given constant.

basic block - A group of atoms or intermediate code which contains no label or branch code.

binary search tree - A connected data structure in which each node has, at most, two links and there are no cycles; it must also have the property that the nodes are ordered, with all of the nodes in the left subtree preceding the node, and all of the nodes in the right subtree following the node.

bison - A public domain version of yacc.

bootstrapping - The process of using a program as input to itself, as in compiler development, through a series of increasingly larger subsets of the source language.

bottom up parsing - Finding the structure of a string in a way that produces or traverses the derivation tree from bottom to top.

byte code - The intermediate form put out by a java compiler.

closure - Another term for the Kleene * operation.

code generation - The phase of the compiler which produces machine language object code from syntax trees or atoms.

comment - Text in a source program which is ignored by the compiler, and is for the programmer's reference only.

compile time - The time at which a program is compiled, as opposed to run time. Also, the time required for compilation of a program.

compiler - A software translator which accepts, as input, a program written in a particular high-level language and produces, as output, an equivalent program in machine language for a particular machine.

compiler-compiler - A program which accepts, as input, the specifications of a programming language and the specifications of a target machine, and produces, as output, a compiler for the specified language and machine.

conflict - In bottom up parsing, the failure of the algorithm to find an appropriate shift or reduce operation.

constant folding - An optimization technique which involves detecting operations on constants, which could be done at compile time rather than at run time.

context-free grammar - A grammar in which the left side of each rule consists of a nonterminal being rewritten (type 2).

context-free language - A language which can be specified by a context-free grammar.

context-sensitive grammar - A grammar in which the left side of each

rule consists of a nonterminal being rewritten, along with left and right context, which may be null (type 1).

context-sensitive language - A language which can be specified by a context-sensitive grammar.

conventional machine language - The language in which a computer architecture can be programmed, as distinguished from a microcode language.

cross compiling - The process of generating a compiler for a new computer architecture, automatically.

DAG - Directed acyclic graph.

data flow analysis - A formal method for tracing the way information about data objects flows through a program, used in optimization.

dead code - Code, usually in an intermediate code string, which can be removed because it has no effect on the output or final results of a program.

derivation - A sequence of applications of rewriting rules of a grammar, beginning with the starting nonterminal and ending with a string of terminal symbols.

derivation tree - A tree showing a derivation for a context-free grammar, in which the interior nodes represent nonterminal symbols and the leaves represent terminal symbols.

deterministic - Having the property that every operation can be completely and uniquely determined, given the inputs (as applied to a machine).

deterministic context-free language - A context-free language which can be accepted by a deterministic pushdown machine.

directed acyclic graph (DAG) - A graph consisting of nodes connected with one-directional arcs, in which there is no path from any node back to itself.

disjoint - Not intersecting.

embedded actions - In a yacc grammar rule, an action which is not at the end of the rule.

empty set - The set containing no elements.

endmarker - A symbol, N, used to mark the end of an input string (used here with pushdown machines).

equivalent grammars - Grammars which specify the same language.

equivalent programs - Programs which have the same input/output relation.

example (of a nonterminal) - A string of input symbols which may be derived from a particular nonterminal.

expression - A language construct consisting of an operation and zero, one, or two operands, each of which may be an object or expression.

extended pushdown machine - A pushdown machine which uses the replace operation.

extended pushdown translator - A pushdown machine which has both an output function and a replace operation.

finite state machine - A theoretical machine consisting of a finite set of states, a finite input alphabet, and a state transition function which specifies the machine's state, given its present state and the current input.

follow set (of a nonterminal, A) - The set of all terminals (or endmarker) which can immediately follow the nonterminal A in a sentential form derived from S.

formal language - A language which can be defined by a precise specification.

front end - The first few phases of the compiler, lexical and syntax analysis, which are machine independent.

global optimization - Improvement of intermediate code in space and/or time.

goto table - A table in LR parsing algorithms which determines which stack symbol is to be pushed when a reduce operation is performed.

grammar - A language specification system consisting of a finite set of rewriting rules involving terminal and nonterminal symbols.

handle - The string of symbols on the parsing stack, which matches the right side of a grammar rule in order for a reduce operation to be performed, in a bottom up parsing algorithm.

hash function - A computation using the value of an item to be stored in a table, to determine the item's location in the table.

hash table - A data structure in which the location of a node's entry is determined by a computation on the node value, called a hash function.

Helper - A macro in SableCC, used to facilitate the definition of tokens.

high-level language - A programming language which permits operations, control structures, and data structures more complex than those available on a typical computer architecture.

identifier - A word in a source program representing a data object, type, or procedure.

Ignored Tokens - The section of a Sablecc specification in which unused tokens may be declared.

implementation language - The language in which a compiler exists.

inherited attributes - Those attributes in an attributed grammar which receive values from nodes on the same or higher levels in the derivation tree.

input alphabet - The alphabet of characters used to make up the strings in a given language.

intermediate form - A language somewhere between the source and object languages.

interpreter - A programming language processor which carries out the intended operations, rather than producing, as output, an object program.

jump over jump optimization - The process of eliminating unnecessary Jump instructions.

keyword - A word in a source program, usually alphanumeric, which has a predefined meaning to the compiler.

language - A set of strings.

left recursion - The grammar property that the right side of a rule begins with the same nonterminal that is being defined by that rule.

left-most derivation - A derivation for a context-free grammar, in which the left-most nonterminal is always rewritten.

lex - A lexical analyzer generator utility in the Unix programming environment which uses regular expressions to define patterns.

lexeme - The output of the lexical analyzer representing a single word in the source program; a lexical token.

lexical analysis - The first phase of the compiler, in which words in the source program are converted to a sequence of tokens representing entities such as keywords, numeric constants, identifiers, operators, etc.

LL(1) grammar - A grammar in which all rules defining the same nonterminal have disjoint selection sets.

LL(1) language - A language which can be described by an LL(1) grammar.

load/store architecture - A computer architecture in which data must be loaded into a CPU register before performing operations.

load/store optimization - The process of eliminating unnecessary Load and Store operations.

local optimization - Optimization applied to object code, usually by examining relatively small blocks of code.

loop invariant - A statement or construct which is independent of, or static within, a particular loop structure.

LR - A class of bottom up parsing algorithms in which the input string is read from the left, and a right-most derivation is found.

LR(k) - An LR parsing algorithm which looks ahead at most k input symbols.

method - In Java, a sub-program with zero or more parameters, belonging to a particular class.

multiple pass code generator - A code generator which reads the the intermediate code string more than once, to handle forward references.

multiple pass compiler - A compiler which scans the source program more than once.

natural language - A language used by people, which cannot be defined perfectly with a precise specification system.

newline - A character, usually entered into the computer as a Return or Enter key, which indicates the end of a line on an output device. Internally, it is usually coded as some combination of 10 and/or 13.

nondeterministic - Not deterministic; i.e., having the property that an input could result in any one of several operations, or that an input could result in no specified operation (as applied to a machine).

nonterminal symbol - A symbol used in the rewriting rules of a grammar, which is not a terminal symbol.

normal form - A method for choosing a unique member of an equivalence class; left-most (or right-most) derivations are a normal form for context-free derivations.

null string - The string consisting of zero characters.

nullable nonterminal - A nonterminal from which the null string can be derived.

nullable rule - A grammar rule which can be used to derive the null string.

object language - The language of the target machine; the output of the compiler is a program in this language.

object program - A program produced as the output of the compiler.

operator - A source language symbol used to specify an arithmetic, assignment, comparison, logical, or other operation involving one or two operands.

optimization - The process of improving generated code in run time and/or space.

p-code - A standard intermediate form developed at the University of California at San Diego.

palindrome - A string which reads the same from left to right as it does from right to left.

parse - A description of the structure of a valid string in a formal language, or to find such a description.

parser - The syntax analysis phase of a compiler.

parsing algorithm - An algorithm which solves the parsing problem for a particular class of grammars.

parsing problem - Given a grammar and an input string, determine whether the string is in the language of the grammar and, if so, find its structure (as in a derivation tree, for example).

pop - A pushdown machine operation used to remove a stack symbol from the top of the stack.

postfix traversal - A tree-scanning algorithm in which the children of a node are visited, followed by the node itself; used to generate object code from a syntax tree.

production - A rewriting rule in a grammar.

Productions - The section of a Sablecc specification in which productions (i.e. grammar rules) are specified.

programming language - A language used to specify a sequence of operations to be performed by a computer.

push - A pushdown machine operation used to place a stack symbol on top of the stack.

pushdown machine - A finite state machine, with an infinite last-in first-out stack; the top stack symbol, current state, and current input are used to determine the next state.

pushdown translator - A pushdown machine with an output function, used to translate input strings into output strings.

quasi-simple grammar - A simple grammar which permits rules rewritten as the null string, as long as the follow set is disjoint with the selection sets of other rules defining the same nonterminal.

quasi-simple language - A language which can be described with a quasi-simple grammar.

recursive descent - A top down parsing algorithm in which there is a procedure for each nonterminal symbol in the grammar.

reduce/reduce conflict - In bottom up parsing, the failure of the algorithm to determine which of two or more reduce operations is to be performed in a particular stack and input configuration.

reduce operation - The operation of replacing 0 or more symbols on the top of the parsing stack with a nonterminal grammar symbol, in a bottom up parsing algorithm.

reduction in strength - The process of replacing a complex operation with an equivalent, but simpler, operation during optimization.

reflexive transitive closure (of a relation) - The relation, R', formed from a given relation, R, including all pairs in the given relation, all reflexive

pairs (a R' a), and all transitive pairs (a R' c if a R' b and b R' c).

register allocation - The process of assigning a purpose to a particular register, or binding a register to a source program variable or compiler variable, so that for a certain range or scope of instructions that register can be used to store no other data.

register-displacement addressing - An address mode in which a complete memory address is formed by adding the contents of a CPU register to the value of the displacement instruction field.

regular expression - An expression involving three operations on sets of strings: union, concatenation, and Kleene * (also known as closure).

relation - A set of ordered pairs.

replace - An extended pushdown machine operation, equivalent to a pop operation, followed by zero or more push operations.

reserved word - A key word which is not available to the programmer for use as an identifier.

rewriting rule - The component of a grammar which specifies how a string of nonterminal and terminal symbols may be rewritten as another string of nonterminals and terminals. Also called a *production*.

right linear grammar - A grammar in which the left side of each rule is a single nonterminal and the right side of each rule is either a terminal or a terminal followed by a nonterminal (type 3).

right linear language - A language which can be specified by a right linear grammar.

right-most derivation - A derivation for a context-free grammar, in which the right-most nonterminal symbol is always the one rewritten.

run time - The time at which an object program is executed, as opposed to compile time.

SableCC - An object-oriented, Java-based compiler generator.

scanner - The phase of the compiler which performs lexical analysis.

selection set - The set of terminals which may be used to direct a top down parser to apply a particular grammar rule.

semantic analysis - That portion of the compiler which generates intermediate code and which attempts to find non-syntactic errors by checking types and declarations of identifiers.

semantics - The intent, or meaning, of an input string.

sentential form - An intermediate form in a derivation which may contain nonterminal symbols.

set - A collection of unique objects.

shift operation - The operation of pushing an input symbol onto the parsing stack, and advancing to the next input symbol, in a bottom up parsing algorithm.

shift reduce parser - A bottom up parsing algorithm which uses a sequence of shift and reduce operations to transform an acceptable input string to the starting nonterminal of a given grammar.

shift/reduce conflict - In bottom up parsing, the failure of the algorithm to determine whether a shift or reduce operation is to be performed in a particular stack and input configuration.

simple algebraic optimization - The elimination of instructions which add 0 to or multiply 1 by a number.

simple grammar - A grammar in which the right side of every rule begins with a terminal symbol, and all rules defining the same nonterminal begin with a different terminal.

simple language - A language which can be described with a simple grammar.

single pass code generator - A code generator which keeps a fixup table for forward references, and thus needs to read the intermediate code string only once.

single pass compiler - A compiler which scans the source program only once.

source language - The language in which programs may be written and used as input to a compiler.

source program - A program in the source language, intended as input to a compiler.

state - A machine's status, or memory/register values. Also, in SableCC, the present status of the scanner.

States - The section of a Sablecc specification in which lexical states may be defined.

starting nonterminal - The nonterminal in a grammar from which all derivations begin.

stdin - In Unix or MSDOS, the standard input file, normally directed to the keyboard.

stdout - In Unix or MSDOS, the standard output file, normally directed to the user.

string - A list or sequence of characters from a given alphabet.

string space - A memory buffer used to store string constants and possibly identifier names or key words.

symbol table - A data structure used to store identifiers and possibly other lexical entities during compilation.

syntax - The specification of correctly formed strings in a language, or the correctly formed programs of a programming language.

syntax analysis - The phase of the compiler which checks for syntax errors in the source program, using, as input, tokens put out by the lexical phase and producing, as output, a stream of atoms or syntax trees.

syntax directed translation - A translation in which a parser or syntax specification is used to specify output as well as syntax.

syntax tree - A tree data structure showing the structure of a source program or statement, in which the leaves represent operands, and the internal nodes represent operations or control structures.

synthesized attributes - Those attributes in an attributed grammar which receive values from lower nodes in the derivation tree.

target machine - The machine for which the output of a compiler is intended.

terminal symbol - A symbol in the input alphabet of a language specified by a grammar.

token - The output of the lexical analyzer representing a single word in the source program.

Tokens - The section of a Sablecc specification in which tokens are defined.

top down parsing - Finding the structure of a string in a way that produces or traverses the derivation tree from top to bottom.

Translation class - An extension of the DepthFirstAdapter class generated by SableCC. It is used to implement actions in a translation grammar.

translation grammar - A grammar which specifies output for some or all input strings.

underlying grammar - The grammar resulting when all action symbols are removed from a translation grammar.

unreachable code - Code, usually in an intermediate code string, which can never be executed.

unrestricted grammar - A grammar in which there are no restrictions on the form of the rewriting rules (type 0).

unrestricted language - A language which can be specified by an unrestricted grammar.

white space - Blank, tab, or newline characters which appear as nothing on an output device.

yacc - (Yet Another Compiler-Compiler) A parser generator utility in the Unix programming environment which uses a grammar to specify syntax.

A

Appendix A - Decaf Grammar

In this appendix, we give a description and grammar of the source language that we call "Decaf." *Decaf* is a simple subset of the standard Java language. It does not include arrays, structs, unions, files, sets, switch statements, do statements, classs definitions, methods, or or many of the low level operators. The only data types permitted are int and float. A complete grammar for Decaf is shown below, and it is similar to the SableCC grammar used in the compiler in Appendix B. Here we use the convention that symbols beginning with uppercase letters are nonterminals, and all other symbols are terminals (i.e., lexical tokens). As in BNF, we use the vertical bar | to indicate alternate definitions for a nonterminal.

Program →	class identifier { public static void main (String[] identifier) CompoundStmt }
Declaration →	Type IdentList ;
Type →	int
	| float
IdentList →	identifier , IdentList
	| identifier
Stmt →	AssignStmt
	| ForStmt
	| WhileStmt
	| IfStmt
	| CompoundStmt
	| Declaration
	| NullStmt
AssignStmt →	AssignExpr ;
ForStmt →	for (OptAssignExpr; OptBoolExpr ; OptAssignExpr) Stmt
OptAssignExpr →	AssignExpr
	|ϵ
OptBoolExpr →	BoolExpr
	|ϵ
WhileStmt →	while (BoolExpr) Stmt
IfStmt →	if (BoolExpr) Stmt ElsePart

ElsePart →	else Stmt
	|ϵ
CompoundStmt→	{ StmtList }
StmtList →	StmtList Stmt
	|ϵ
NullStmt →	;
BoolExpr →	Expr Compare Expr
Compare →	==
	| <
	| >
	| <=
	| >=
	|! =
Expr →	AssignExpr
	| Rvalue
AssignExpr →	identifier = Expr
Rvalue →	Rvalue + Term
	| Rvalue - Term
	| Term
Term →	Term * Factor
	| Term / Factor
	| Factor
Factor →	(Expr)
	| - Factor
	| + Factor
	| identifier
	| number

This grammar is used in Appendix B as a starting point for the SableCC grammar for our Decaf compiler, with some modifications. It is not unusual for a compiler writer to make changes to the given grammar (which is descriptive of the source language) to obtain an equivalent grammar which is more amenable for parsing. Decaf is clearly a very limited programming language, yet despite its limitations it can be used to program some useful applications. For example, a Decaf program to compute the cosine function is shown in Figure A.1.

```
class AClass {
public static void main (String[] args)
{float cos, x, n, term, eps, alt;
// compute the cosine of x to within tolerance eps
// use an alternating series
      x = 3.14159;
      eps = 0.1;
      n = 1;
      cos = 1;
      term = 1;
      alt = -1;
      while (term>eps)
            {
            term = term * x * x / n / (n+1);
            cos = cos + alt * term;
            alt = -alt;
            n = n + 2;
            }
}
}
```

Figure A.1: Decaf program to compute the cosine function using a Taylor series

Appendix B - Decaf Compiler

B.1 Installing Decaf

The compiler for Decaf shown in this appendix is implemented using SableCC. The bottom-up parser produces a syntax tree, which when visited by the Translation class, puts out a file of atoms, which forms the input to the code generator, written as a separate C program. The code generator puts out hex bytes to stdout (the standard output file), one instruction per line. This output can be displayed on the monitor, stored in a file, or piped into the Mini simulator and executed.

This software is available in source code from the author via the Internet. The file names included in this package are, at the time of this printing:

decaf.grammar	Grammar file, input to SableCC
Translation.java	Class to implement a translation from syntax tree to atoms
Compiler.java	Class containing a main method, to invoke the parser and translator.
Atom.java	Class to define an atom
AtomFile.java	Class to define the file storing atoms
gen.c	Code generator
mini.c	Target machine simulator
mini.h	Header file for simulator
miniC.h	Header file for simulator
cos.decaf	Decaf program to compute the cosine function
bisect.decaf	Decaf program to compute the square root of two by bisection
exp.decaf	Decaf program to compute ex.
fact.decaf	Decaf program to compute the factorial function
compile	Script to compile a decaf program and write code to stdout
compileAndGo	Script to compile a decaf program and execute the resulting code with the mini simulator.

The source files are available at: `http://cs.rowan.edu/~bergmann/books/java/decaf`

These are all plain text files, so you should be able to simply choose File | Save As from your browser window. Create a subdirectory named decaf, and download the files *.java to the decaf subdirectory. Download all other files into your current directory (i.e. the parent of decaf).

To build the Decaf compiler, there are two steps (from the directory containing decaf.grammar). First generate the parser, lexer, analysis, and node classes (the exact form of this command could depend on how SableCC has been installed on your system):

$ sablecc decaf.grammar

The second step is to compile the java classes that were not generated by SableCC:

$ javac decaf/*.java

You now have a Decaf compiler; the main method is in decaf/Compiler.class. To compile a decaf program, say cos.decaf, invoke the compiler, and redirect stdin to the decaf source file:

$ java decaf.Compiler < cos.decaf

This will create a file named atoms, which is the the result of translating cos.decaf into atoms. To create machine code for the mini architecture, and execute it with a simulator, you will need to compile the code generator, and the simulator, both of which are written in standard C:

```
$   cc  gen.c -o gen
$   cc  mini.c -o mini
```

Now you can generate mini code, and execute it. Simply invoke the code generator, which reads from the file atoms, and writes mini instructions to stdout. You can pipe these instructions into the mini machine simulator:

$ gen | mini

The above can be simplified by using the scripts provided. To compile the file cos.decaf, without executing, use the compile script:

$ compile cos

To compile and execute, use the compileAndGo script:

$ compileAndGo cos

This software was developed and tested using a Sun V480 running Solaris 10. The reader is welcome to adapt it for use on Windows and other systems. A flow graph indicating the relationships of these files is shown in Figure B.2 in which input and output file names are shown in rectangles. The source files are shown in Appendix B.2, below, with updated versions available at `http://cs.rowan.edu/~bergmann/books`.

B.2 Source Code for Decaf

In this section we show the source code for the Decaf compiler. An updated version may be obtained at http://cs.rowan.edu/∼bergmann/books The first source file is the grammar file for Decaf, used as input to SableCC. This is described in section 5.5.

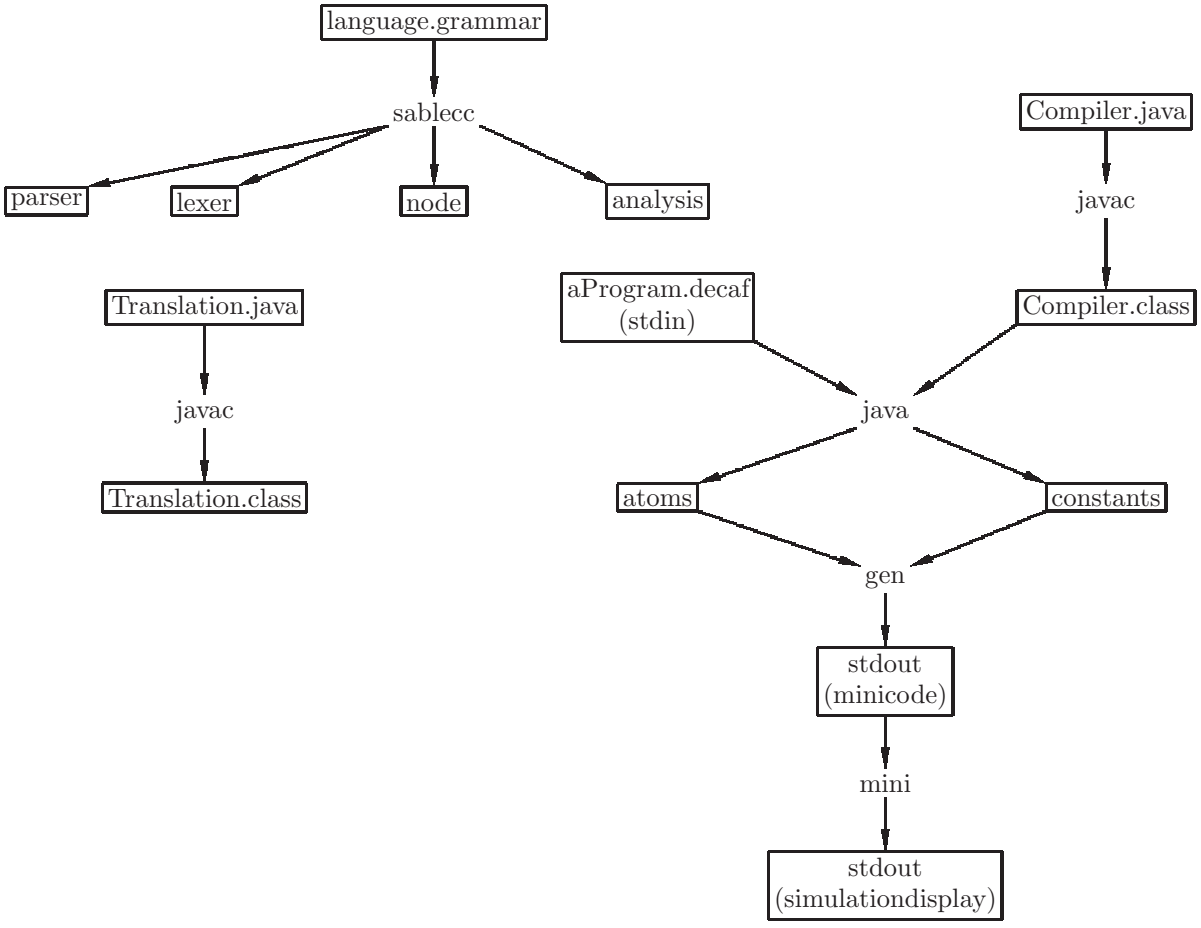


Figure B.2: Flow Diagram to Compile and Execute a Decaf Program

```
//       decaf.grammar
// SableCC grammar for decaf, a subset of Java.
// March 2003,  sdb

Package decaf;

Helpers                                 // Examples
  letter = ['a'..'z'] | ['A'..'Z'] ; //   w
  digit =    ['0'..'9'] ; //   3
  digits =   digit+ ; // 2040099
  exp  =     ['e' + 'E'] ['+' + '-']? digits;   // E-34
  newline = [10 + 13]   ;
  non_star = [[0..0xffff] - '*'];
  non_slash = [[0..0xffff] - '/'];
  non_star_slash = [[0..0xffff] - ['*' + '/']];


Tokens
  comment1 = '//' [[0..0xffff]-newline]* newline ;
  comment2 = '/*' non_star* '*'
 (non_star_slash non_star* '*'+)* '/' ;

  space = ' ' | 9 | newline ; // '\t'=9   (tab)
  clas = 'class' ; // key words (reserved)
  public = 'public' ;
  static = 'static' ;
  void = 'void' ;
  main = 'main' ;
  string = 'String' ;
  int = 'int' ;
  float = 'float' ;
  for = 'for' ;
  while = 'while' ;
  if = 'if' ;
  else = 'else' ;
  assign = '=' ;
  compare = '==' | '<' | '>' | '<=' | '>=' | '!=' ;
  plus = '+' ;
  minus = '-' ;
  mult = '*' ;
  div = '/' ;
  l_par = '(' ;
  r_par = ')' ;
  l_brace = '{' ;
  r_brace = '}' ;
  l_bracket = '[' ;
```

```
  r_bracket = ']' ;
  comma = ',' ;
  semi = ';' ;
  identifier = letter (letter | digit | '_')* ;
  number  =  (digits '.'? digits? | '.'digits) exp? ;
// Example:  2.043e+5
  misc = [0..0xffff] ;

Ignored Tokens
  comment1, comment2, space;


Productions

  program =   clas identifier l_brace public static
void main l_par string l_bracket
r_bracket [arg]: identifier r_par
compound_stmt r_brace ;
  type =                  {int}    int
                        | {float}  float ;
  declaration =         type identifier identlist* semi;
  identlist =           comma identifier ;

  stmt =    {dcl}      declaration
  | {stmt_no_trlr}  stmt_no_trailer
  | {if_st} if_stmt
| {if_else_st} if_else_stmt
| {while_st} while_stmt
| {for_st} for_stmt
;
  stmt_no_short_if = {stmt_no_trlr} stmt_no_trailer
           | {if_else_no_short}    if_else_stmt_no_short_if
           | {while_no_short}      while_stmt_no_short_if
           | {for_no_short}        for_stmt_no_short_if
                        ;
  stmt_no_trailer =   {compound}    compound_stmt
| {null}        semi
| {assign}      assign_stmt
;

  assign_stmt =         assign_expr semi
;

  for_stmt = for l_par assign_expr? semi bool_expr?
  [s2]: semi [a2]: assign_expr? r_par stmt    ;
  for_stmt_no_short_if = for l_par assign_expr? semi      bool_expr? [s2]: semi [a2]:
```

```
    assign_expr? r_par
  stmt_no_short_if
  ;

  while_stmt =    while l_par bool_expr r_par stmt
;
  while_stmt_no_short_if = while l_par bool_expr r_par
stmt_no_short_if
;

  if_stmt = if l_par bool_expr r_par stmt
;
  if_else_stmt = if l_par bool_expr r_par stmt_no_short_if  else stmt
;
  if_else_stmt_no_short_if = if l_par bool_expr r_par  [if1]: stmt_no_short_if else
[if2]: stmt_no_short_if
;

  compound_stmt = l_brace stmt* r_brace
;

  bool_expr = expr compare [right]: expr
;

  expr =   {assn} assign_expr
| {rval} rvalue
;
  assign_expr =  identifier assign expr ;
  rvalue =          {plus}  rvalue plus term
| {minus} rvalue minus term
| {term}  term
;
  term =          {mult}  term mult factor
                | {div}   term div factor
                | {fac}   factor
                ;
  factor =        {pars}   l_par expr r_par
                | {uplus}  plus factor
                | {uminus} minus factor
                | {id}     identifier
                | {num}    number
                ;
```

The file Translation.java is the Java class which visits every node in the syntax tree and produces a file of atoms and a file of constants. It is described

in section 5.5.

```
//         Translation.java
// Translation class for decaf, a subset of Java.
// Output atoms from syntax tree
//   sdb  March 2003
//   sdb  updated May 2007
//     to use generic maps instead of hashtables.

package decaf;
import decaf.analysis.*;
import decaf.node.*;
import java.util.*;
import java.io.*;

class Translation extends DepthFirstAdapter
{

// All stored values are doubles, key=node, value is memory loc or label number
Map  <Node, Integer> hash = new HashMap <Node, Integer> (); // May 2007

Integer zero = new Integer (0);
Integer one  = new Integer (1);

AtomFile out;

////////////////////////////////////////////////
// Definition of Program

public void inAProgram (AProgram prog)
//   The class name and main args need to be entered into symbol table
//     to avoid error message.
//   Also, open the atom file for output
{  identifiers.put (prog.getIdentifier().toString(), alloc()); // class name
   identifiers.put (prog.getArg().toString(), alloc()); // main (args)
   out = new AtomFile ("atoms");
}

public void outAProgram (AProgram prog)
//  Write the run-time memory values to a file "constants".
//  Close the binary file of atoms so it can be used for
//    input by the code generator
{  outConstants();
   out.close();
}
```

```
////////////////////////////////////////////////
// Definitions of declaration and identlist

public void inADeclaration (ADeclaration node)
{   install (node.getIdentifier()); }

public void outAIdentlist (AIdentlist node)
{   install (node.getIdentifier()); }

void install (TIdentifier id)
//  Install id into the symbol table
{  Integer loc;
   loc = identifiers.get (id.toString());
   if (loc==null)
identifiers.put (id.toString(), alloc());
   else
System.err.println ("Error: " + id + " has already been declared ");
}


////////////////////////////////////////////////
// Definition of for_stmt

public void caseAForStmt (AForStmt stmt)
{  Integer lbl1, lbl2, lbl3;
   lbl1 = lalloc();
   lbl2 = lalloc();
   lbl3 = lalloc();
   inAForStmt (stmt);
   if (stmt.getFor() !=null) stmt.getFor().apply(this);
   if (stmt.getLPar() !=null) stmt.getLPar().apply(this);
   if (stmt.getAssignExpr()!=null) // initialize
       { stmt.getAssignExpr().apply(this);
atom ("LBL", lbl1);
}
   if (stmt.getSemi() != null) stmt.getSemi().apply(this);
   if (stmt.getBoolExpr() != null) // test for termination
{ stmt.getBoolExpr().apply(this);
atom ("JMP", lbl2);
atom ("LBL", lbl3);
}
   if (stmt.getS2() != null) stmt.getS2().apply(this);
   if (stmt.getA2() != null)
{ stmt.getA2().apply(this); // increment
atom ("JMP", lbl1);
```

```
atom ("LBL", lbl2);
}
   if (stmt.getRPar() != null) stmt.getRPar().apply(this);
   if (stmt.getStmt() != null)
{ stmt.getStmt().apply(this);
atom ("JMP", lbl3);
atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
}
   outAForStmt(stmt);
}

public void caseAForStmtNoShortIf (AForStmtNoShortIf stmt)
{  Integer lbl1, lbl2, lbl3;
   lbl1 = lalloc();
   lbl2 = lalloc();
   lbl3 = lalloc();
   inAForStmtNoShortIf (stmt);
   if (stmt.getFor() !=null) stmt.getFor().apply(this);
   if (stmt.getLPar() !=null) stmt.getLPar().apply(this);
   if (stmt.getAssignExpr()!=null) // initialize
       { stmt.getAssignExpr().apply(this);
atom ("LBL", lbl1);
}
   if (stmt.getSemi() != null) stmt.getSemi().apply(this);
   if (stmt.getBoolExpr() != null) // test for termination
{ stmt.getBoolExpr().apply(this);
atom ("JMP", lbl2);
atom ("LBL", lbl3);
}
   if (stmt.getS2() != null) stmt.getS2().apply(this);
   if (stmt.getA2() != null)
{ stmt.getA2().apply(this); // increment
atom ("JMP", lbl1);
atom ("LBL", lbl2);
}
   if (stmt.getRPar() != null) stmt.getRPar().apply(this);
   if (stmt.getStmtNoShortIf() != null)
{ stmt.getStmtNoShortIf().apply(this);
atom ("JMP", lbl3);
atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
}
   outAForStmtNoShortIf (stmt);
}

////////////////////////////////////////////////
// Definition of while_stmt
```

```
public void inAWhileStmt (AWhileStmt stmt)
{  Integer lbl = lalloc();
   hash.put (stmt, lbl);
   atom ("LBL", lbl);
}

public void outAWhileStmt (AWhileStmt stmt)
{  atom ("JMP", (Integer) hash.get(stmt));
   atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
}

public void inAWhileStmtNoShortIf (AWhileStmtNoShortIf stmt)
{  Integer lbl = lalloc();
   hash.put (stmt, lbl);
   atom ("LBL", lbl);
}

public void outAWhileStmtNoShortIf (AWhileStmtNoShortIf stmt)
{  atom ("JMP", (Integer) hash.get(stmt));
   atom ("LBL", (Integer) hash.get (stmt.getBoolExpr()));
}


//////////////////////////////////////////
// Definition of if_stmt

public void outAIfStmt (AIfStmt stmt)
{  atom ("LBL", (Integer) hash.get (stmt.getBoolExpr())); } // Target for bool_expr's TST

// override the case of if_else_stmt
public void caseAIfElseStmt (AIfElseStmt node)
{  Integer lbl = lalloc();
   inAIfElseStmt (node);
   if (node.getIf() != null) node.getIf().apply(this);
   if (node.getLPar() != null) node.getLPar().apply(this);
   if (node.getBoolExpr() != null)node.getBoolExpr().apply(this);
   if (node.getRPar() != null) node.getRPar().apply(this);
   if (node.getStmtNoShortIf() != null)
     {  node.getStmtNoShortIf().apply(this);
    atom ("JMP", lbl); // Jump over else part
    atom ("LBL", (Integer) hash.get (node.getBoolExpr()));
     }
   if (node.getElse() != null) node.getElse().apply(this);
   if  (node.getStmt() != null) node.getStmt().apply(this);
   atom ("LBL", lbl);
```

```
   outAIfElseStmt (node);
}

// override the case of if_else_stmt_no_short_if
public void caseAIfElseStmtNoShortIf (AIfElseStmtNoShortIf node)
{  Integer lbl = lalloc();
   inAIfElseStmtNoShortIf (node);
   if (node.getIf() != null) node.getIf().apply(this);
   if (node.getLPar() != null) node.getLPar().apply(this);
   if (node.getBoolExpr() != null)node.getBoolExpr().apply(this);
   if (node.getRPar() != null) node.getRPar().apply(this);
   if (node.getIf1() != null)
     { node.getIf1().apply(this);
    atom ("JMP", lbl); // Jump over else part
    atom ("LBL", (Integer) hash.get (node.getBoolExpr()));
     }
   if (node.getElse() != null) node.getElse().apply(this);
   if  (node.getIf2() != null) node.getIf2().apply(this);
   atom ("LBL", lbl);
   outAIfElseStmtNoShortIf (node);
}

////////////////////////////////////////////////
// Definition of bool_expr

public void outABoolExpr (ABoolExpr node)
{  Integer lbl = lalloc();
   hash.put (node, lbl);
   atom ("TST", (Integer) hash.get(node.getExpr()),
(Integer) hash.get(node.getRight()),
zero,
// Negation of a comparison code is 7 - code.
new Integer (7  - getComparisonCode (node.getCompare().toString())),
lbl);
}


////////////////////////////////////////////
// Definition of expr

public void outAAssnExpr (AAssnExpr node)
// out of alternative {assn} in expr
{ hash.put (node, hash.get (node.getAssignExpr())); }

public void outARvalExpr (ARvalExpr node)
// out of alternative {rval} in expr
```

```
{  hash.put (node, hash.get (node.getRvalue())); }


int getComparisonCode (String cmp)
//  Return the integer comparison code for a comparison
{   if (cmp.indexOf ("==")>=0) return  1;
    if (cmp.indexOf ("<")>=0)  return  2;
    if (cmp.indexOf (">")>=0)  return  3;
    if (cmp.indexOf ("<=")>=0) return  4;
    if (cmp.indexOf (">=")>=0) return  5;
    if (cmp.indexOf ("!=")>=0) return  6;
  return  0; // this should never occur
}

////////////////////////////////////////////////
// Definition of assign_expr

public void outAAssignExpr (AAssignExpr node)
//  Put out the MOV atom
{  Integer assignTo = getIdent (node.getIdentifier());
   atom ("MOV", (Integer) hash.get (node.getExpr()),
 zero,
 assignTo);
   hash.put (node, assignTo);
}


////////////////////////////////////////////////
// Definition of rvalue

public void outAPlusRvalue (APlusRvalue node)
{// out of alternative {plus} in Rvalue, generate an atom ADD.
    Integer i = alloc();
    hash.put (node, i);
    atom ("ADD", (Integer)hash.get(node.getRvalue()),
         (Integer) hash.get(node.getTerm()) , i);
}

public void outAMinusRvalue(AMinusRvalue node)
{// out of alternative {minus} in Rvalue, generate an atom SUB.
    Integer i = alloc();
    hash.put (node, i);
    atom ("SUB", (Integer) hash.get(node.getRvalue()),
 (Integer) hash.get(node.getTerm()), i);
}
```

```
public void outATermRvalue (ATermRvalue node)
//  Attribute of the rvalue is the same as the term.
{   hash.put (node, hash.get (node.getTerm())); }


////////////////////////////////////////////
// Definition of term

public void outAMultTerm (AMultTerm node)
{// out of alternative {mult} in Term, generate an atom MUL.
    Integer i = alloc();
    hash.put (node, i);
    atom ("MUL", (Integer)hash.get(node.getTerm()),
         (Integer) hash.get(node.getFactor()) , i);
}

public void outADivTerm(ADivTerm node)
{// out of alternative {div} in Term, generate an atom DIV.
    Integer i = alloc();
    hash.put (node, i);
    atom ("DIV", (Integer) hash.get(node.getTerm()),
 (Integer) hash.get(node.getFactor()), i);
}

public void outAFacTerm (AFacTerm node)
{ //  Attribute of the term is the same as the factor
hash.put (node, hash.get(node.getFactor()));
 }


Map <Double, Integer> nums = new HashMap <Double, Integer> ();
Map <String, Integer > identifiers = new HashMap <String, Integer> ();

final int MAX_MEMORY = 1024;
Double memory [] = new Double [MAX_MEMORY];
int memHigh = 0;
// No, only memory needs to remain for codegen.


// Maintain a hash table of numeric constants, to avoid storin
g//   the same number twice.
// Move the number to a run-time memory location.
// That memory location will be the attribute of the Number token.
public void caseTNumber(TNumber num)
{ Integer loc;
  Double dnum;
```

```
  // The number as a Double
  dnum = new Double (num.toString());
  // Get its memory location
  loc = (Integer) nums.get (dnum);
  if (loc==null) // Already in table?
     { loc = alloc(); // No, install in table of nums
nums.put (dnum, loc);
// Store value in run-time memory
memory[loc.intValue()] = dnum;
// Retain highest memory loc
   if (loc.intValue() > memHigh)
   memHigh = loc.intValue();
     }
  hash.put (num, loc); // Set attribute to move up tree
}


Integer getIdent(TIdentifier id)
// Get the run-time memory location to which this id is bound
{  Integer loc;
   loc = identifiers.get (id.toString());
   if (loc==null)
System.err.println ("Error: " + id +
" has not been declared");
   return loc;
}

////////////////////////////////////////////
// Definition of factor

public void outAParsFactor (AParsFactor node)
{   hash.put (node, hash.get (node.getExpr())); }

// Unary + doesn't need any atoms to be put out.
public void outAUplusFactor (AUplusFactor node)
{   hash.put (node, hash.get (node.getFactor())); }

// Unary - needs a negation atom (NEG).
public void outAUminusFactor (AUminusFactor node)
{   Integer loc = alloc(); // result of negation
    atom ("NEG", (Integer)hash.get(node.getFactor()), zero, loc);
    hash.put (node, loc);
}

public void outAIdFactor (AIdFactor node)
{   hash.put (node, getIdent (node.getIdentifier())); }
```

```
public void outANumFactor (ANumFactor node)
{   hash.put (node, hash.get (node.getNumber())); }


////////////////////////////////////////////////////////////////
//  Send the run-time memory constants to a file for use by the code generator.

void outConstants()
{  FileOutputStream fos = null;
   DataOutputStream ds = null;
   int i;

   try
    { fos = new FileOutputStream ("constants");
ds = new DataOutputStream (fos);
    }
   catch (IOException ioe)
    { System.err.println ("IO error opening constants file for output: "
+ ioe);
    }

   try
    {  for (i=0; i<=memHigh ; i++)
    if (memory[i]==null) ds.writeDouble (0.0); // a variable is bound here
    else
     ds.writeDouble (memory[i].doubleValue());
    }
   catch (IOException ioe)
    { System.err.println ("IO error writing to constants file: "
+ ioe);
    }
   try { fos.close(); }
   catch (IOException ioe)
    { System.err.println ("IO error closing constants file: "
+ ioe);
    }
}


//////////////////////////////////////////////////////
// Put out atoms for conversion to machine code.
// These methods display to stdout, and also write to a
//   binary file of atoms suitable as input to the code generator.

void atom (String atomClass, Integer left, Integer right, Integer result)
```

```
{   System.out.println (atomClass +  " T" + left + " T"  + right +  " T" +
result);
    Atom atom = new Atom (atomClass, left, right, result);
    atom.write(out);
}

void atom (String atomClass, Integer left, Integer right, Integer result,
Integer cmp, Integer lbl)
{   System.out.println (atomClass +  " T" + left + " T"  + right +  " T" +
result + " C" + cmp + " L" + lbl);
    Atom atom = new Atom (atomClass, left, right, result, cmp, lbl);
    atom.write(out);
}

void atom (String atomClass, Integer lbl)
{   System.out.println (atomClass +  " L" + lbl);
    Atom atom = new Atom (atomClass, lbl);
    atom.write(out);
}

static int avail = 0;
static int lavail = 0;

Integer alloc()
{ return new Integer (++avail); }

Integer lalloc()
{ return new Integer (++lavail); }


}
```

The file Compiler.java defines the Java class which contains a main method which invokes the parser to get things started. It is described in section 5.5.

```
//       Compiler.java
//  main method which invokes the parser and reads from
//     stdin
//  March 2003   sdb

package decaf;
import decaf.parser.*;
import decaf.lexer.*;
import decaf.node.*;
```

```
import java.io.*;

public class Compiler
{

public static void main(String[] arguments)
{ try
  { System.out.println();

    // Create a Parser instance.
    Parser p = new Parser
        ( new Lexer
          ( new PushbackReader
            ( new InputStreamReader(System.in),
   1024)));

    // Parse the input.
    Start tree = p.parse();

    // Apply the translation.
    tree.apply(new Translation());

    System.out.println();
  }
  catch(Exception e)
    { System.out.println(e.getMessage()); }
}
}
```

The file Atom.java defines the Java class Atom, which describes an Atom and permits it to be written to an output file. It is described in section 5.5.

```
//     Atom.java
// Define an atom for output by the Translation class

package decaf;
import java.io.*;

class Atom
// Put out atoms to a binary file in the format expected by
//  the old code generator.
{
static final int ADD = 1;
static final int SUB = 2;
```

```
static final int MUL = 3;
static final int DIV = 4;
static final int JMP = 5;
static final int NEG = 10;
static final int LBL = 11;
static final int TST = 12;
static final int MOV = 13;

int cls;
int left;
int right;
int result;
int cmp;
int lbl;

// constructors for Atom
Atom (String cls, Integer l, Integer r, Integer res)
{   setClass (cls);
    left = l.intValue();
    right = r.intValue();
    result = res.intValue();
    cmp = 0;
    lbl = 0;
}

Atom (String cls, Integer l, Integer r, Integer res,
Integer c, Integer lb)
{   setClass (cls);
    left = l.intValue();
    right = r.intValue();
    result = res.intValue();
    cmp = c.intValue();
    lbl = lb.intValue();
}

Atom (String cls,  Integer lb)
{   setClass (cls);
    left = 0;
    right = 0;
    result = 0;
    cmp = 0;
    lbl = lb.intValue();
}
}

void setClass (String c)
```

```
//  set the atom class as an int code
{   if (c.equals("ADD")) cls = ADD;
    else if (c.equals("SUB")) cls = SUB;
    else if (c.equals("MUL")) cls = MUL;
    else if (c.equals("DIV")) cls = DIV;
    else if (c.equals("JMP")) cls = JMP;
    else if (c.equals("NEG")) cls = NEG;
    else if (c.equals("LBL")) cls = LBL;
    else if (c.equals("TST")) cls = TST;
    else if (c.equals("MOV")) cls = MOV;
}

void write (AtomFile out)
 // write a single atom out to the binary file
{
    try
     {  out.ds.writeInt (cls);
        out.ds.writeInt (left);
        out.ds.writeInt (right);
        out.ds.writeInt (result);
        out.ds.writeInt (cmp);
        out.ds.writeInt (lbl);
     }
    catch (IOException ioe)
     {
        System.out.println
("IO Error writing to atom file, atom class is "
                + cls + ", error is " + ioe);
     }
}
}
```

The file AtomFile.java defines the Java class AtomFile, which permits output of an Atom to a file.

```
//         AtomFile.java
//  Create the binary output file for atoms
//  March 2003   sdb

package decaf;
import java.io.*;

class AtomFile
{
```

```
   FileOutputStream fos;
   DataOutputStream ds;
   String fileName;


AtomFile (String name)
{  fileName = new String (name);
   try
     {  fos = new FileOutputStream (fileName);
        ds = new DataOutputStream (fos);
     }
   catch (IOException ioe)
     {
        System.err.println ("IO error opening atom file
(" + fileName + "): " + ioe);
     }
}

void close()
{   try
     { ds.close(); }
    catch (IOException ioe)
     {  System.err.println ("IO error closing atom file
(" + fileName + "): " + ioe);
     }


}
}
```

B.3 Code Generator

The code generator is written in the C language; we re-use the code generator written for the C/C++ version of this book, and it is stored in the file gen.c. This program reads from a file named 'atoms' and a file named 'constants' which are produced by the Translation class from the syntax tree. It writes instructions in hex characters for the Mini machine simulator to stdout, the standard output file. This can be displayed on the monitor, stored in a file, or piped directly into the Mini simulator as described.

The code generator output also includes a hex location and disassembled op code on each line. These are ignored by the Mini machine simulator and are included only so that the student will be able to read the output and understand how the compiler works.

The first line of output is the starting location of the program instructions.

Program variables and temporary storage are located beginning at memory location 0, consequently the Mini machine simulator needs to know where the first instruction is located. The function out_mem() sends the constants which have been stored in the target machine memory to stdout. The function dump_atom() is included for debugging purposes only; the student may use it to examine the atoms produced by the parser.

The code generator solves the problem of forward jump references by making two passes over the input atoms. The first pass is implemented with a function named build_labels() which builds a table of Labels (a one dimensional array), associating a machine address with each Label.

The file of atoms is closed and reopened for the second pass, which is implemented with a switch statement on the input atom class. The important function involved here is called gen(), and it actually generates a Mini machine instruction, given the operation code (atom class codes and corresponding machine operation codes are the same whenever possible), register number, memory operand address (all addressing is absolute), and a comparison code for compare instructions. Register allocation is kept as simple as possible by always using floating-point register 1, and storing all results in temporary locations. The source code for the code generator, from the file gen.c, is shown below. For an updated version of this source code, see `http://cs.rowan.edu/~bergmann/books`.

```
/*      gen.c
   Code generator for mini architecture.
  Input should be a file of atoms, named "atoms"

*********************************************************
   Modified March 2003 to work with decaf as well as miniC.
   sdb
*/
#define NULL 0
#include "mini.h"
#include "miniC.h"

struct atom inp;
long labels[MAXL];
ADDRESS pc=0;
int ok = TRUE;
long lookup (int);

/* code_gen ()  */
void main()     /* March 2003, for Java.  sdb  */
{ int r;

  /* send target machine memory containing constants to stdout */
```

```
/*  end_data = alloc(0);     March 2003   sdb
    constants precede instructions */
/* send run-time memory  constants to stdout */
  out_mem();

  atom_file_ptr = fopen ("atoms","rb");  /* open file of  atoms */
  pc = end_data;     /* starting address of instructions */
  build_labels();  /* first pass */
  fclose (atom_file_ptr);

  /* open file of atoms for */
  atom_file_ptr = fopen ("atoms","rb");
  get_atom();  /* second pass */
  pc = end_data;
  ok = TRUE;
 while (ok)
  {
 /*  dump_atom();  */ /* for debugging, etc */

  switch (inp.op)  /* check atom class */
        { case ADD:  gen (LOD, r=regalloc(),inp.left);
                     gen (ADD, r, inp.right);
                     gen (STO, r, inp.result);
                break;
        case SUB:    gen (LOD, r=regalloc(), inp.left);
                     gen (SUB, r, inp.right);
                     gen (STO, r, inp.result);
                break;
        case NEG:    gen (CLR, r=regalloc());
                     gen (SUB, r, inp.left);
                     gen (STO, r, inp.result);
                break;
        case MUL:    gen (LOD, r=regalloc(), inp.left);
                     gen (MUL, r, inp.right);
                     gen (STO, r, inp.result);
                break;
        case DIV:    gen (LOD, r=regalloc(), inp.left);
                     gen (DIV, r, inp.right);
                     gen (STO, r, inp.result);
                break;
        case JMP:    gen (CMP, 0, 0, 0);
                     gen (JMP);
                break;
        case TST:     gen (LOD, r=regalloc(), inp.left);
                      gen (CMP, r, inp.right, inp.cmp);
                      gen (JMP);
```

```
                break;
        case MOV:     gen (LOD, r=regalloc(), inp.left);
                      gen (STO, r, inp.result);
                break;
        }
  get_atom();
 }
  gen (HLT);
}


get_atom()
/* read an atom from the file of atoms into inp */
/* ok indicates that an atom was actually read */
{ int n;

  n = fread (&inp, sizeof (struct atom), 1, atom_file_ptr);
  if (n==0) ok = FALSE;
}

dump_atom()
{ printf ("op: %d  left: %04x  right: %04x  result: %04x  cmp: %d  dest: %d\n",
           inp.op, inp.left, inp.right, inp.result, inp.cmp, inp.dest); }

gen (int op, int r, ADDRESS add, int cmp)
/* generate an instruction to stdout
   op is the simulated machine operation code
   r is the first operand register
   add is the second operand address
   cmp is the comparison code for compare instructions
1 is ==
      2 is <
      3 is >
      4 is <=
      5 is >=
      6 is !=
   jump destination is taken from the atom inp.dest
*/
{union {struct fmt instr;
        unsigned long word;
       } outp;
 outp.word = 0;

 outp.instr.op = op; /* op code */
 if (op!=JMP)
   { outp.instr.r1 = r; /* first operand */
```

```
     outp.instr.s2 = add; /* second operand */
   }
  else outp.instr.s2 = lookup (inp.dest);
/* jump destination */
 if (op==CMP) outp.instr.cmp = cmp;
/* comparison code 1-6 */

 printf ("%08x\t%04x\t%s\n", outp.word, pc, op_text(op));
 pc++;
}


int regalloc ()
/* allocate a register for use in an instruction */
{ return 1; }


build_labels()
/* Build a table of label values on the first pass */
{
  get_atom();
  while (ok)
  {
  if (inp.op==LBL)
        labels[inp.dest] = pc;

        /* MOV and JMP atoms require two instructions,
           all other atoms require three instructions. */
        else if (inp.op==MOV || inp.op==JMP) pc += 2;
                else pc += 3;
  get_atom();
  }
}


long lookup (int label_num)
/* look up a label in the table and return it's memory address */
{ return labels[label_num];
}


out_mem()
/* send target machine memory contents to stdout.  this is the beginning of the object file, to b
{
  ADDRESS i;
  data_file_ptr = fopen ("constants","rb");
```

```
/* open file of constants
   March 2003   sdb*/
  get_data();


  printf ("%08x\tLoc\tDisassembled Contents\n", end_data);
               /* starting address of instructions */
  for (i=0; i<end_data; i++)
        printf ("%08x\t%04x\t%8lf\n", memory[i].instr, i,
                memory[i].data);
}


void get_data()
/* March 2003  sdb
   read a number from the file of constants into inp_const
   and store into memory.
*/
{ int i,status=1;
  double inp_const;
  for (i=0; status; i++)
    {   status = fread (&inp_const, sizeof (double), 1, data_file_ptr);
        memory[i].data = inp_const ;
    }
  end_data = i-1;
}


char * op_text(int operation)
/* convert op_codes to mnemonics */
{
  switch (operation)
        { case CLR: return "CLR";
          case ADD: return "ADD";
          case SUB: return "SUB";
          case MUL: return "MUL";
          case DIV: return "DIV";
          case JMP: return "JMP";
          case CMP: return "CMP";
          case LOD: return "LOD";
          case STO: return "STO";
          case HLT: return "HLT";
        }
}
```

Appendix C - Mini Simulator

The Mini machine simulator is simply a C program stored in the file mini.c. It reads instructions and data in the form of hex characters from the standard input file, stdin. The instruction format is as specified in Section 6.5.1, and is specified with a structure called fmt in the header file, mini.h.

The simulator begins by calling the function boot(), which loads the Mini machine memory from the values in the standard input file, stdin, one memory location per line. These values are numeric constants, and zeroes for program variables and temporary locations. The boot() function also initializes the program counter, PC (register 1), to the starting instruction address.

The simulator fetches one instruction at a time into the instruction register, ir, decodes the instruction, and performs a switch operation on the operation code to execute the appropriate instruction. The user interface is designed to allow as many instruction cycles as the user wishes, before displaying the machine registers and memory locations. The display is accomplished with the dump() function, which sends the Mini CPU registers, and the first sixteen memory locations to stdout so that the user can observe the operation of the simulated machine. The memory locations displayed can be changed easily by setting the two arguments to the dumpmem() function. The displays include both hex and decimal displays of the indicated locations.

As the user observes a program executing on the simulated machine, it is probably helpful to watch the memory locations associated with program variables in order to trace the behavior of the original MiniC program. Though the compiler produces no memory location map for variables, their locations can be determined easily because they are stored in the order in which they are declared, beginning at memory location 3. For example, the program that computes the cosine function begins as shown here:

```
... public static void main (String [] args)
{  float cos, x, n, term, eps, alt;
```

In this case, the variables cos, x, n, term, eps, and alt will be stored in that

order in memory locations 3 through 8.

The source code for the Mini machine is in the file mini.c and is shown below:

```
/* simulate the Mini architecture */
/* 32-bit word addressable machine, with 16 general registers and
      16 floating-point registers.
   r1:  program counter
   ir:      instruction register
   r0-r15:      general registers (32 bits)
   fpr0-fpr15:      floating-point registers (32 bits)

   instruction format:
      bits      function
      0-3      opcode   1      r1 = r1+s2
                        2      r1 = r1-s2
                        4      r1 = r1*s2
                        5      r1 = r1/s2
                        7      pc = S2 if flag       JMP
                        8      flag = r1 cmp s2      CMP
                        9      r1 = s2               Load
                        10      s2 = r1              Store
                        11      r1 = 0               Clear

      4        mode    0      s2 is 20 bit address
                       1      s2 is 4 bit reg (r2) and 16 bit offset (o2)

      5-7      cmp     0      always true
                       1      ==
                       2      <
                       3      >
                       4      <=
                       5      >=
                       6      !=

      8-11     r1            register address for first operand
      12-31    s2            storage adress if mode=0
      12-15    r2            part of storage address if mode=1
      16-31    o2            rest of storage address if mode=1
                              if mode=1, s2 = c(r2) + o2 */

#include <stdio.h>
#include mini.h
#define PC reg[1]

FILE * tty;                          /* read from keyboard */
```

```
unsigned long addr;
unsigned int flag, r2, o2;

main ()
{
int n = 1, count;

boot(); /* load memory from stdin */

tty = fopen (/dev/tty, r);      /* read from keyboard                                    e
                                       redirected */
while (n>0)
{
for (count = 1; count<=n; count++)
  {                                    /* fetch */
      ir.full32 = memory[PC++].instr;
      if (ir.instr.mode==1)
         {  o2 = ir.instr.s2 & 0x0ffff;
            r2 = ir.instr.s2 & 0xf0000;
            addr = reg[r2] + o2;}
      else  addr = ir.instr.s2;

      switch (ir.instr.op)
        { case ADD:      fpreg[ir.instr.r1].data =  fpreg[ir.instr.r1].data +
                        memory[addr].data;
                 break;
          case SUB:      fpreg[ir.instr.r1].data =  fpreg[ir.instr.r1].data -
                        memory[addr].data;
                 break;
          case MUL:      fpreg[ir.instr.r1].data = fpreg[ir.instr.r1].data *
                        memory[addr].data;
                 break;
          case DIV:      fpreg[ir.instr.r1].data = fpreg[ir.instr.r1].data /
                        memory[addr].data;
                 break;
             case JMP:      if (flag) PC = addr;  /* conditional jump */
                 break;
          case CMP:      switch (ir.instr.cmp)
                   {case 0:      flag = TRUE;      /* unconditional */
                        break;
                 case 1:      flag = fpreg[ir.instr.r1].data == memory[addr].data;
                        break;
                 case 2:      flag = fpreg[ir.instr.r1].data < memory[addr].data;
                        break;
```

```
                  case 3:      flag = fpreg[ir.instr.r1].data > memory[addr].data;
                        break;
                  case 4:      flag = fpreg[ir.instr.r1].data <= memory[addr].data;

                 break;
                  case 5:      flag = fpreg[ir.instr.r1].data >= memory[addr].data;
                        break;
                  case 6:      flag = fpreg[ir.instr.r1].data != memory[addr].data;
                  }
      case LOD:      fpreg[ir.instr.r1].data  = memory[addr].data;
            break;
      case STO: memory[addr].data = fpreg[ir.instr.r1].data;
            break;
      case CLR: fpreg[ir.instr.r1].data = 0.0;
             break;
      case HLT:  n = -1;
      }
  }
   dump ();
   printf ("Enter number of instruction cycles, 0 for no change, or -1 to quit\n");
      /* read from keyboard if stdin is redirected */
   fscanf (tty,"%d", &count);
   if (count!=0 && n>0) n = count;
}
}

void dump ()
{ dumpregs();
  dumpmem(0,15);
}

void dumpregs ()
{int i;
 char * pstr;

 printf ("ir = %08x\n", ir.full32);
 for (i=0; i<8; i++)
   { if (i==1) pstr = PC = ; else pstr =      ;
     printf ("%s reg[%d] = %08x = %d\tfpreg[%d] = %08x = %e\n",
      pstr,i,reg[i],reg[i],i,fpreg[i].instr,fpreg[i].data);
   }
}

void dumpmem(int low, int high)
{int i;
 char * f;
```

```
 low = low/4*4;
 high = (high+4)/4*4 - 1;
 if (flag) f = "TRUE"; else f = "FALSE";
 printf ("memory\t\t\t\tflag = %s\naddress\t\tcontents\n",f);
 for (i=low; i<=high; i+=4)
   printf ("%08x\t%08x     %08x     %08x     %08x\n\t\t%8e %8e %8e %8e\n",
            i,memory[i].instr,memory[i+1].instr,memory[i+2].instr,
            memory[i+3].instr, memory[i].data, memory[i+1].data,
            memory[i+2].data,
      memory[i+3].data);
}

void boot()
/* load memory from stdin */
{ int i = 0;

  scanf ("%8lx%*[^\n]\n", &PC);            /* starting address of instructions */

  while (EOF!=scanf ("%8lx%*[^\n]\n", &memory[i++].instr));
}
```

The only source files that have not been displayed are the header files. The file miniC.h contains declarations, macros, and includes which are needed by the compiler but not by the simulator. The file mini.h contains information needed by the simulator.

The header file miniC.h is shown below:

```
/* Size of hash table for identifier symbol table */
#define HashMax 100

/* Size of table of compiler generated address labels */
#define MAXL 1024

/* memory address type on the simulated machine */
typedef unsigned long ADDRESS;

/* Symbol table entry */
struct Ident
{char * name;
struct Ident * link;
int type; /* program name = 1,
     integer = 2,
     real = 3 */
ADDRESS memloc;};
```

```
/* Symbol table */
struct Ident * HashTable[HashMax];

/* Linked list for declared identifiers */
struct idptr
{struct Ident * ptr;
 struct idptr * next;
};
struct idptr * head = NULL;
int dcl = TRUE; /* processing the declarations section */

/* Binary search tree for numeric constants */
struct nums
{ADDRESS memloc;
struct nums * left;
struct nums * right;};
struct nums *  numsBST = NULL;

/* Record for file of atoms */
struct atom
{int op; /* atom classes are shown below */
ADDRESS left;
ADDRESS right;
ADDRESS result;
int cmp; /* comparison codes are 1-6 */
int dest;
};

/* ADD, SUB, MUL, DIV, and JMP are also atom classes */
/* The following atom classes are not op codes */
#define NEG 10
#define LBL 11
#define TST 12
#define MOV 13

FILE * atom_file_ptr;
ADDRESS avail = 0, end_data = 0;
int err_flag = FALSE; /* has an error been detected? */
```

The header file mini.h is shown below:

```
#define MaxMem 0xffff
#define TRUE 1
```

```
#define FALSE 0

/* Op codes are defined here: */
#define CLR 0
#define ADD 1
#define SUB 2
#define MUL 3
#define DIV 4
#define JMP 5
#define CMP 6
#define LOD 7
#define STO 8
#define HLT 9

/* Memory word on the simulated machine may be treated as numeric data or as an instruction */
union { float data;
unsigned long instr;
} memory [MaxMem];


/* careful!  this structure is machine dependent! */
struct fmt
  { unsigned int s2:    20;
    unsigned int r1: 4;
    unsigned int cmp:    3;
    unsigned int mode:   1;
    unsigned int op: 4;
   }
   ;

union {
struct fmt instr;
unsigned long full32;
      } ir;

unsigned long reg[8];
union { float data;
unsigned long instr;
       } fpreg[8];
```

Bibliography

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2007.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[3] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[4] Ken Arnold and James Gosling. *The Java Programming Lanuguage*. Addison Wesley, 1996.

[5] William A. Barrett, Rodney M. Bates, David A. Gustafson, and John D. Couch. *Compiler Construction: Theory and Practice*. Science Research Associates, 1979.

[6] Bill Campbell, Swami lyer, and Bahar Akbal-Delibas. *Introduction to Compiler Construction in a Java World*. CRC, 2013.

[7] Noam Chomsky. Certain formal properties of grammars. *Information and Control*, 2(2):137–167, June 1958.

[8] Noam Chomsky. *Syntactic Structures*. Mouton, 1965.

[9] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2012.

[10] Etienne Gagnon. Sablecc, an object oriented comp;iler framework. Master's thesis, McGill University. available at http://www.sablecc.org.

[11] Jim Holmes. *Object-Oriented Compiler Construction*. Prentice Hall, 1995.

[12] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[13] Bruce Hutton. Language implementation. unpublished lecture notes, University of Auckland, 1987.

[14] Samuel N. Kamin. *Programming Languages: An Interpreter Based Approach*. Addison Wesley, 1990.

[15] Brian W. Kernighan and Rob Pike. *The Unix Programming Environment.* Prentice Hall, 1984.

[16] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, Dec 1965.

[17] Thomas W. Parsons. *Introduction to Compiler Construction.* Freeman, 1992.

[18] David A. Patterson and John L. Hennessy. *Computer Organization and Design.* Morgan Kaufmann, 2012.

[19] Anthony J. Dos Reiss. *Compiler Construction using Java, JavaCC, and Yacc.* Wiley, 2012.

[20] Sablecc. http://www.sablecc.org.

[21] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice Hall, 1990.

[22] Niklaus Wirth. *Compiler Construction.* Addison Wesley Longman, 1996.

Index

Index