Compiler Design:

Theory, Tools, and Examples

C/C++ Edition

Seth D. Bergmann

Rowan University

2010

Table of 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 (formerly Glassboro State College), 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 an earlier edition that was written for a Pascal based curriculum. As many computer science departments have moved to C++ 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.

The most essential prerequisites for this book are courses in C or C++ 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 C which I call MiniC. Chapters 2, 4, 5, and 6 include a section devoted to explaining how the relevant part of the MiniC 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 MiniC 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/miniC`

Once you have stored the programs on your computer, the programs can be generated with the makefile on unix/linux:

```
> make
```

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 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 me using Adobe Pagemaker, and diagrams were drawn with Microsoft Excel and Powerpoint.

Finally, I am most grateful to Sue, Aaron, and Sarah, for being so understanding during the time that I spent working on this project.

Seth D. Bergmann
bergmann@rowan.edu

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 $((x-x_0)^2 + (x-x_1)^2)^{1/2}$, 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 C++ 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(a) (see p. 3). Why didn't it 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

Sample Problem 1.1 (a)

Show the output of a C/C++ compiler, in any typical assembly language, for the following C/C++ 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:
```

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.

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 don't 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 doesn't 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 language, but

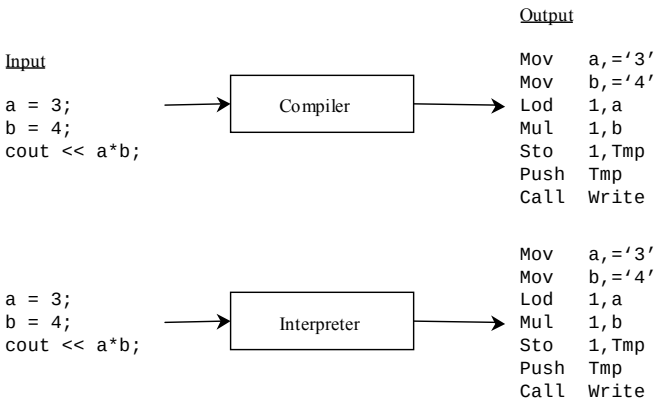


Figure 1.1 A Compiler and Interpreter Producing Very Different Output for the Same Input

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.

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

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

if x<b fn1();
   else fn2();
```

Run-Time Errors

```
x = a-a;
y = 100/x;    // division by 0

ptr = NULL;
data = ptr->info;
// use of null pointer
```

Sample Problem 1.1 (b)

Show the compiler output and the interpreter output for the following C++ source code:

```
for (i=1; i<=4; i++) cout << i*3;
```

Solution:

Compiler

```
    LOD     R1,='4'
    STO     R1,Temp1
    MOV     i,='1'
L1:CMP      i,Temp1
    BH      L2      {Branch if i>Temp1}
    LOD     R1,i
    MUL     R1,='3'
    STO     R1,Temp2
    PUSH    Temp2
    CALL    Write
    ADD     i,='1'  {Add 1 to i}
    B       L1
L2:
```

Interpreter

```
3  6  9  12
```

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[1986] and is shown in Figure 1.2 (see p. 6). 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.

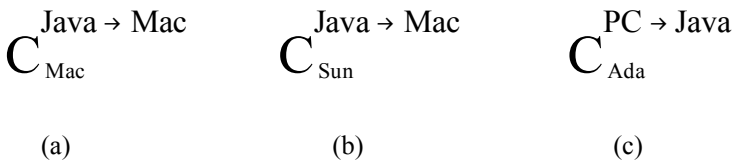


Figure 1.2 Big C notation for compilers: (a) A Java compiler that runs on the Mac (b) A Java compiler that generates Mac programs and runs on a Sun computer (c) A compiler that translates PC programs into Java and is written in Ada.

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 (c)

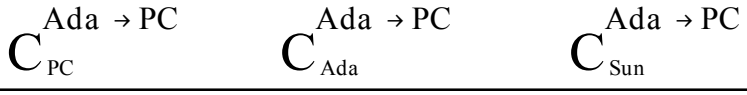
Using the big C notation of Figure 1.2, show each of the following compilers:

(a) An Ada compiler which runs on the PC and compiles to the PC machine language.

(b) An Ada compiler which compiles to the PC machine language, but which is written in Ada.

(c) An Ada compiler which compiles to the PC machine language, but runs on a Sun.

Solution:



Exercises 1.1

1. Show *assembly language* for a machine of your choice, corresponding to each of the following C/C++ 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;
cout <<C<<A<<B;
```

(b)
```
A = 12;
B = 6;
if (A<B) cout << A;
     else cout << B;
```

(c)
```
A = 12;
B = 6;
while (B<A)
     {  A = A-1;
        cout << A << B << endl;
     }
```

3. Which of the following C/C++ 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) then A = 2;
       else A = X;
```

(c)
```
if (A>0) X = 20;
     else if (A<0) X = 10;
          else X = X/A;
```

(d)
```
while ((p->info>0) && (p!=0))
     p = p->next;
/*    assume p points to a struct
      named node with
      these field definitions:
       int info;
       node * next; */
```

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:

(1) ***key words*** - `while`, `void`, `if`, `for`, ...
(2) ***identifiers*** - declared by the programmer
(3) ***operators*** - `+`, `-`, `*`, `/`, `=`, `==`, ...
(4) ***numeric constants*** - numbers such as `124`, `12.35`, `0.09E-23`, etc.
(5) ***character constants*** - single characters or strings of characters enclosed in quotes.
(6) ***special characters*** - characters used as delimiters such as `.` `(` `)` `,` `;` `:`
(7) ***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.

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,

Sample Problem 1.2 (a)

Show the token classes, or "words", put out by the lexical analysis phase corresponding to this C++ 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)
```

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 (b), 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.

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

Sample Problem 1.2 (b)

Show atoms corresponding to the following C/C++ statement:

```
A = B + C * D ;
```

Solution:

```
(MULT,C,D,TEMP1)
(ADD,B,TEMP1,TEMP2)
(MOVE,TEMP2,A)
```

Sample Problem 1.2 (c)

Show atoms corresponding to the C/C++ 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)
```

label `L1`. The atom `(TEST, A, <=, B, L2)` would be a conditional branch to the label `L2`, if `A<=B` is true.

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, below.

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.

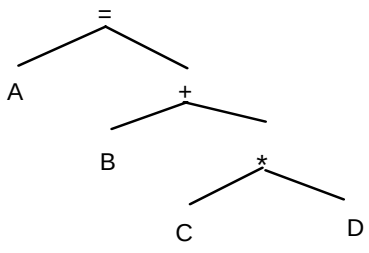


Figure 1.3 A Syntax Tree for `A = B + C * D`

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.

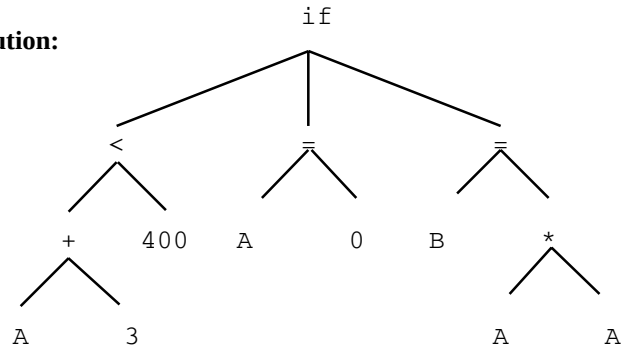
Sample Problem 1.2 (d)

Show a syntax tree for the C/C++ 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 = sqrt (y);        // square root function
      cout << x+i << endl;
   }
```

In this case, the assignment to `x` need not be inside the loop since `y` doesn't 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 = sqrt (y);                  // loop invariant
for (i=1; i<=100000; i++)
      cout << x+i << endl;
```

This would eliminate 99,999 unnecessary calls to the `sqrt` function 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

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 (e), below, 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.

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.

Sample Problem 1.2 (e)

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
```

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 *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, at right. 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 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***.

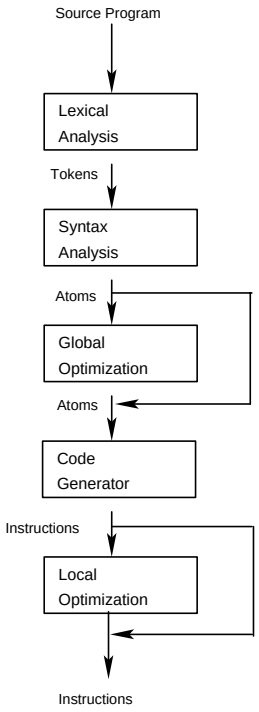


Figure 1.4 The Phases of a Compiler

Exercises 1.2

1. Show the *lexical tokens* corresponding to each of the following C/C++ 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 C/C++ 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 *C/C++ 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 *C/C++ 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)
```
     x = 2;
     goto L99;
     x = 3;
L99: cout << x;
```

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 *C/C++ 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
        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.

In order to describe these implementation techniques graphically, we use the method shown below, 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.

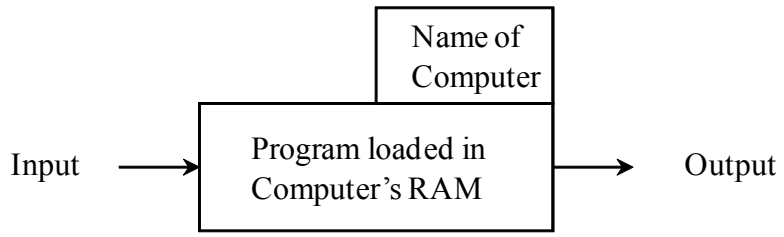


Figure 1.5 Notation for a Program Running on a Computer

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, below. 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.

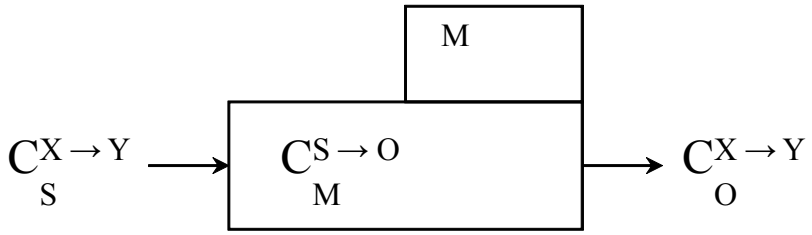
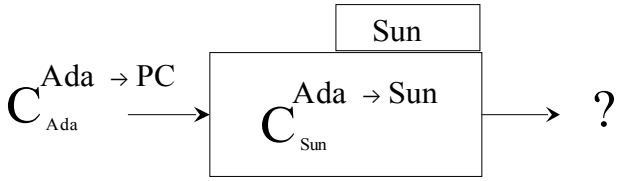


Figure 1.6 Notation for a compiler being translated to a different language

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

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



Solution:

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

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 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 (see p. 21). 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

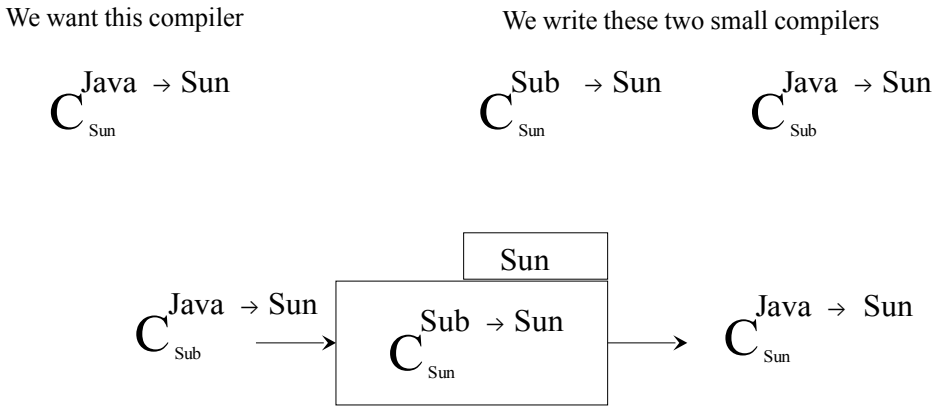


Figure 1.7 Bootstrapping Java onto a Sun Computer

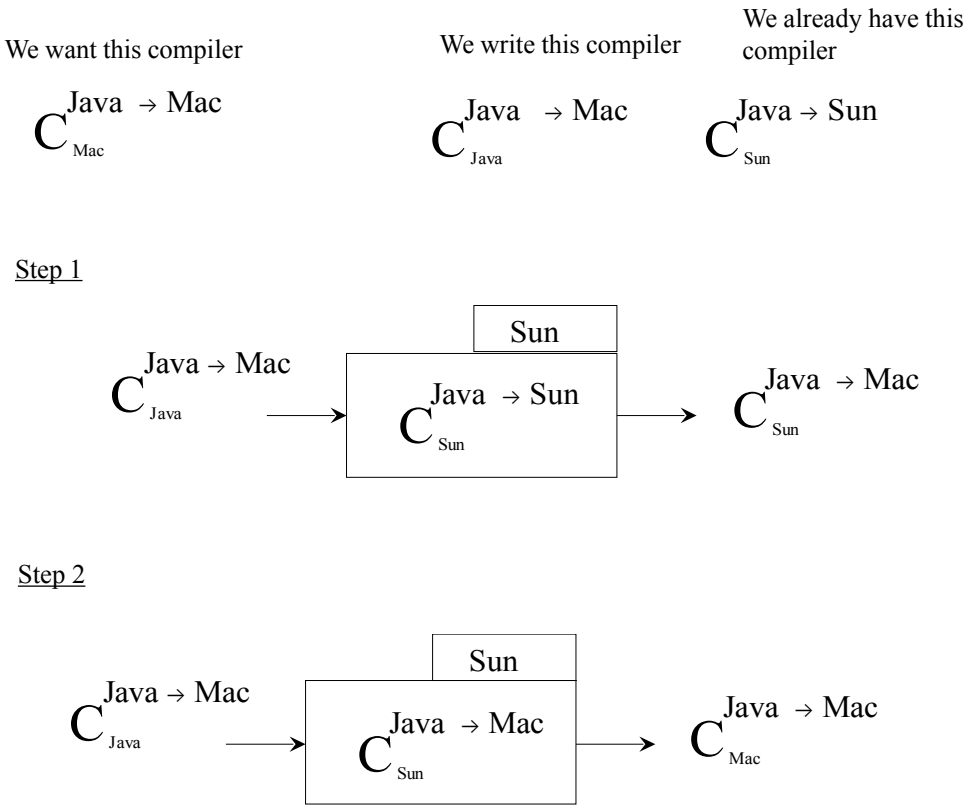


Figure 1.8 Cross compiling Java from a Sun to a Mac computer

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 (see p. 21). 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 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 (see p. 23), 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 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

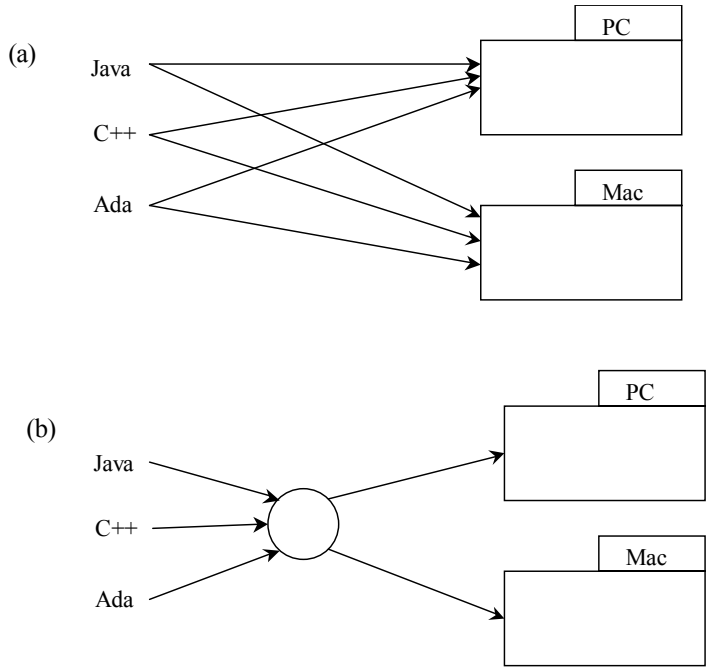


Figure 1.9 (a) Six compilers needed for three languages on two machines (b) Fewer than three compilers using intermediate form needed for the same languages and machines

intermediate form. The Java Virtual Machine (i.e. Java byte code) is another intermediate form which has been used extensively on the Intenet.

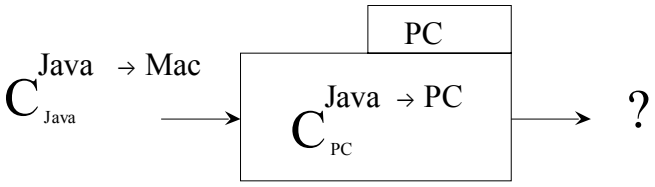
1.3.4 Compiler-Compilers

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 *lex* and *yacc* utilities of the Unix programming environment.

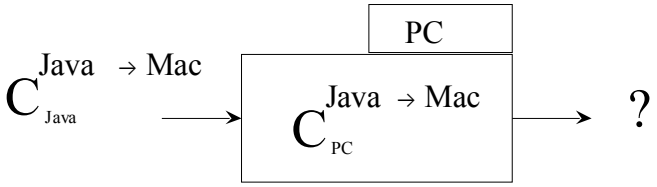
Exercises 1.3

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

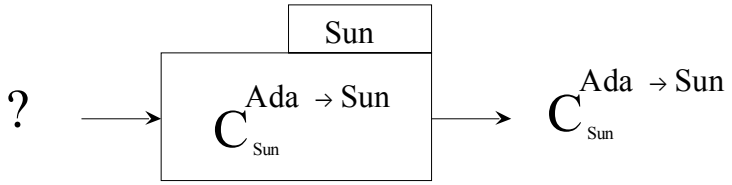
(a)



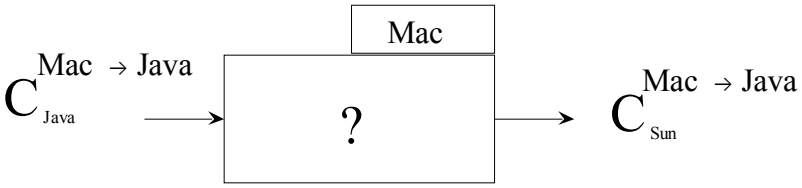
(b)



(c)



(d)



2. How could the compiler generated in part (d) of Question 1 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 FORTRRAN 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, using first a small subset of Ada, `Sub1`, and then a larger subset, `Sub2`. First use `Sub1` to implement `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:

$$C^{\text{Java} \rightarrow \text{FORT}}_{\text{Mac}} \qquad C^{\text{FORT} \rightarrow \text{Java}}_{\text{Sun}} \qquad C^{\text{Java} \rightarrow \text{Sun}}_{\text{Mac}} \qquad C^{\text{Java} \rightarrow \text{FORT}}_{\text{Java}}$$

6. In Figure 1.8 (see p. 21) suppose we also have $C^{\text{Java} \rightarrow \text{Sun}}_{\text{Java}}$. When we write

$$C^{\text{Java} \rightarrow \text{Mac}}_{\text{Java}}$$

, which of the phases of $C^{\text{Java} \rightarrow \text{Sun}}_{\text{Java}}$ 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: MiniC

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 ***MiniC*** as a relatively simple subset of the C language. The implementation of MiniC 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 MiniC.

MiniC is a "bare bones" version of C. Its only data types are `int` and `float`, and it does not permit arrays, structures, enumerated types, or subprograms. However, it does include `while, for`, and `if` control structures, and it is possible to write some useful programs in MiniC. The example that we will use for the case study is the following MiniC program, to compute the cosine function:

```
void cosine()
{ 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:

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

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 MiniC, 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 MiniC 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 MiniC 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 MiniC language, the compiler, or the Mini machine architecture. Some of the exercises in later chapters are designed with this intent.

Exercises 1.4

1. Which of the following are valid program segments in MiniC? Like C, MiniC programs are free-format (Refer to Appendix A).

(a)
```
for (x = 1; x<10; )
     y = 13;
```

(b)
```
if (a<b) { x =
     2;  y = 3 ;}
```

(c)
```
while (a+b==c) if (a!=c)
     a = a + 1;
```

(d)
```
{
     a = 4 ;
     b = 2; ;
}
```

(e)
```
for (i==22; i++; i=3) ;
```

2. Modify the MiniC description given in Appendix A to include a `switch` statement as defined in standard C.

3. Modify the MiniC description given in Appendix A to include a `do while` statement as defined in standard C.

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 *MiniC*, a small subset of the C language, 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 2.1.

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 Pascal 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. Pascal syntax
4. Italian syntax

The third example is the syntax of a *programming language* (in which each string in the language is a Pascal 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 lan-

guage. One solution to this problem is to use a 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) – and returns as result a state.

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 these) will cause the machine to be in an accepting

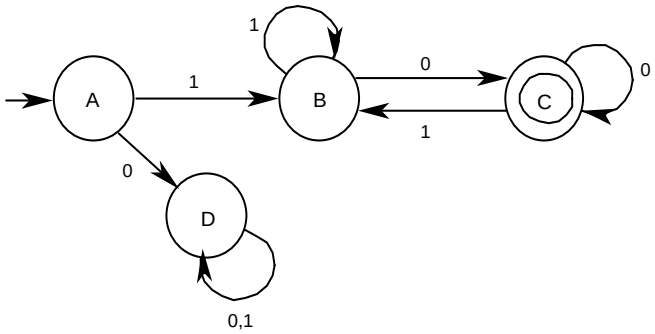


Figure 2.1 Example of a Finite State Machine

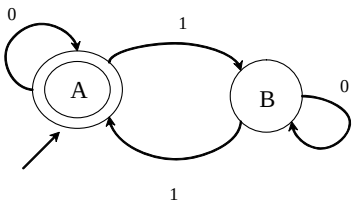


Figure 2.2 Even Parity Checker

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.

	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.

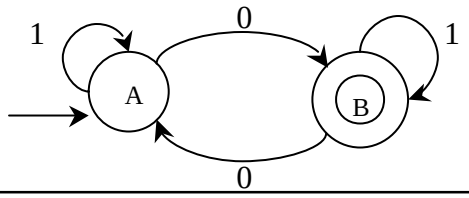
Sample Problem 2.0 (a)

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

Solution:

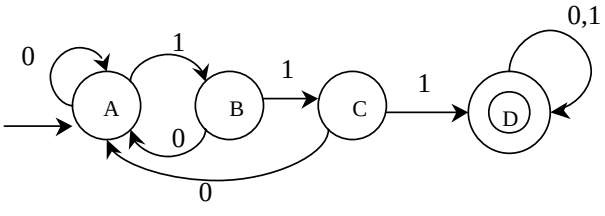
	0	1
A	B	A
*B	A	B



2. Strings containing three consecutive ones

Solution:

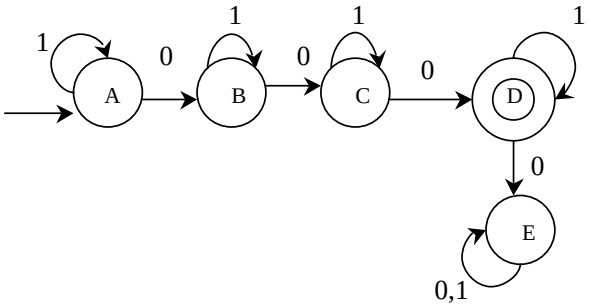
	0	1
A	A	B
B	A	C
C	A	D
*D	D	D



3. Strings containing exactly three zeros

Solution:

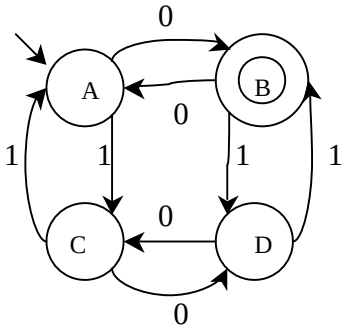
	0	1
A	B	A
B	C	B
C	D	C
*D	E	D
E	E	E



4. Strings containing an odd number of zeros and an even number of ones

Solution:

	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: $s \cdot \varepsilon = s$. 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} ·{b, ε} = {ab·b, ab·ε, a·b, a·ε, c·b, c·ε}
= {abb, ab, a, cb, c}

In this example, the string ab need not be listed twice. Note that if L_1 and L_2 are two languages, then $L_1 \cdot L_2$ is not necessarily equal to $L_2 \cdot L_1$. Also, $L \cdot \{\varepsilon\} = L$, 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:

$L^0 = \{\varepsilon\}$
$L^1 = L$
$L^2 = L \cdot L$

$L^n = L \cdot L^{n-1}$

$L^* = L^0 + L^1 + L^2 + L^3 + L^4 + L^5 + \ldots$

Note that $\phi^* = \{\varepsilon\}$. 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:

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

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 L_1, L_2 and L_3 are languages, then:

$L_1 + L_2 \cdot L_3 = L_1 + (L_2 \cdot L_3)$
$L_1 \cdot L_2^* = L_1 \cdot (L_2^*)$

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^*:

$L^0 = \{\varepsilon\}$
$L^1 = \{0,1\}$
$L^2 = L \cdot L^1 = \{00,01,10,11\}$
$L^3 = L \cdot L^2 = \{000,001,010,011,100,101,110,111\}$

$L^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, \ldots\}$
= the set of all strings of zeros and ones.

Another example:

$1 \cdot (0+1)^* \cdot 0 = 1(0+1)^*0$
= {10, 100, 110, 1000, 1010, 1100, 1110, ...}
= 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·(L·L) = (L·L)·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 (b)

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

Solution:

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

Exercises 2.0

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?

Sample Problem 2.0 (c)

Give a regular expression for each of the languages described in Sample Problem 2.0 (a)

Solutions:

```
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
```

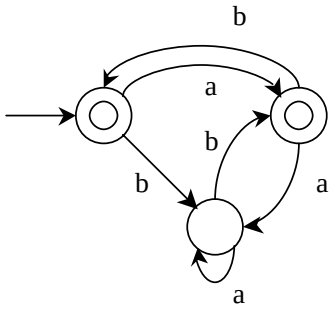
An algorithm for converting a finite state machine to an equivalent regular expression is beyond the scope of this text, but may be found in Hopcroft & Ullman [1979].

(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) Pascal
(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 you 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.

(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.

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) ***keywords*** - `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 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 C++ 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 Class	Token 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.

Exercises 2.1

1. For each of the following C/C++ input strings show the *word boundaries* and *token classes* 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 C/C++ are 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 C++ 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"
```

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:

```
bool accept[STATES];
{
int fsm[STATES] [INPUTS];     // state transition table
char inp;                     // input symbol (8-bit int)
int state = 0;                // starting state;

      while (cin >> inp)
            state = fsm[state] [inp];
}

if (accept[state]) cout << "Accepted";
else cout << "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. 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 C++ 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.

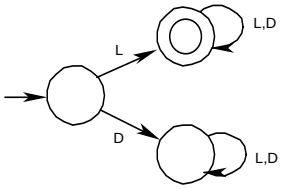


Figure 2.4 Finite State Machine to Accept Identifiers

A third example of the use of state machines in lexical analysis is shown in Figure 2.6. This machine accepts keywords ***if, int, inline, 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.

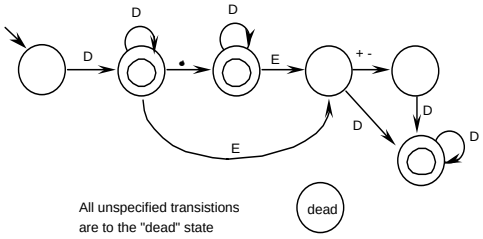


Figure 2.5 A Finite State Machine to Accept Numeric Constants

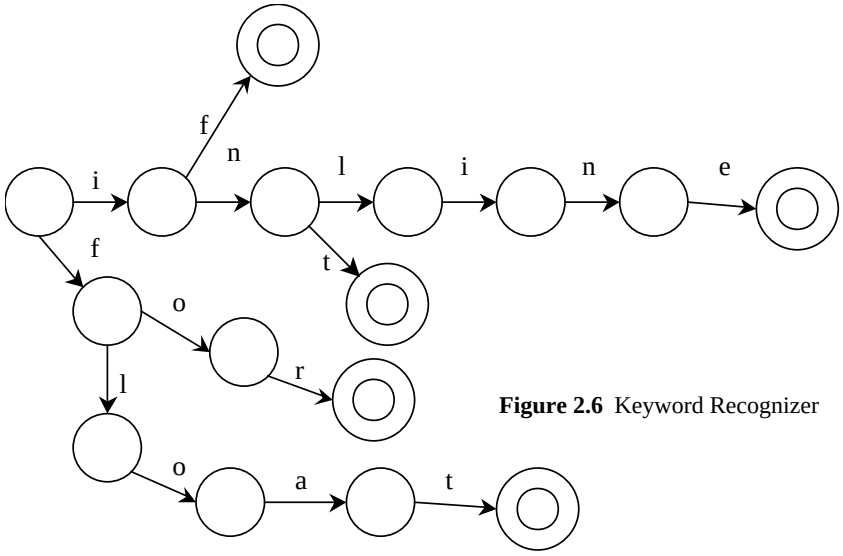


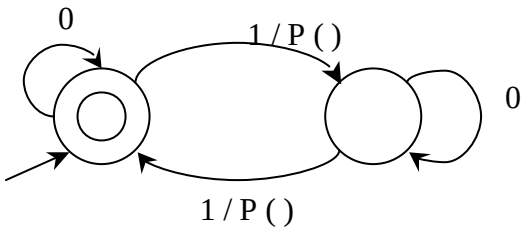
Figure 2.6 Keyword Recognizer

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, below, 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()`.




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

Figure 2.7 Parity Bit Generator

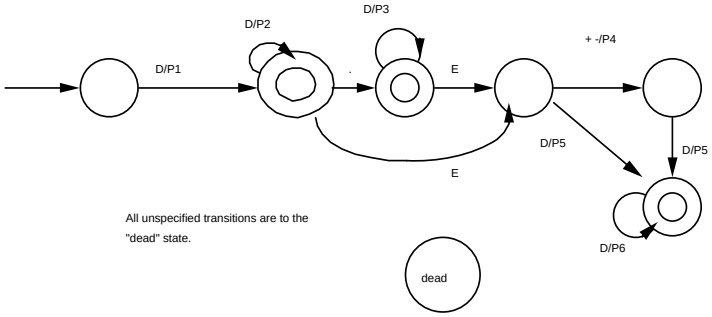
Sample Problem 2.2

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

Solution:

In the state diagram shown below, we have included function calls designated `P1()`, `P2()`, `P3()`, ... 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 function

`P()` before changing state and reading the next input symbol. The functions referred to in the state diagram are shown below:




```
int Places, N, D, Exp, Sign;  // global variables

void P1()
{
      Places = 0;        //Places after decimal point
      N = D;             // Input symbol is a numeric digit
      Exp = 0;           // Default exponent of 10 is 0
      Sign = +1;         // Default sign of exponent is
                         // positive
}
void P2()
{
      N = N*10 + D;      // Input symbol is a numeric digit
}

void P3()
{
      N = N*10 + D;      // Input symbol is a numeric digit
                         // after a decimal point
      Places = Places + 1;  // Count decimal places
}

void P4()
{
      if (input=='-') then sign = -1;  // sign of exponent
}
```

```
void P5()
{
      Exp = D;     // Input symbol is a numeric digit in the
                   // exponent


void P6()
{
      Exp = Exp*10 + D; // Input symbol is a numeric
                        // digit in the Exponent
}
```

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

```
Result = N * Power (10, Sign*Exp  - Places);
```

where Power (x,y) = x^y

Exercises 2.2

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 (see p. 45) 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. 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}.

4. Add *actions* to your solution to Problem 2 so that numeric constants will be computed as in Sample Problem 2.2.

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):

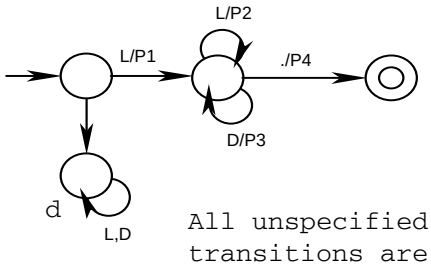
```
int sum;

void P1()                    Void P2()
{                            {
   sum = L;                     sum += L;
}                            }

Void P3()                    int Hash (int n)
{                            {
   sum += D;                    return n % 10;
}                            }

Void P4()
{
   cout << Hash (sum);
}
```

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



All unspecified transitions are to state d.

6. Show the *values* that will be asigned to the variable `N` in Sample Problem 2.2 (see p. 46) 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 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 $O(n \log_2 n)$ 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. Later in the course we will see how the symbol table is used and what additional information is stored in it.

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

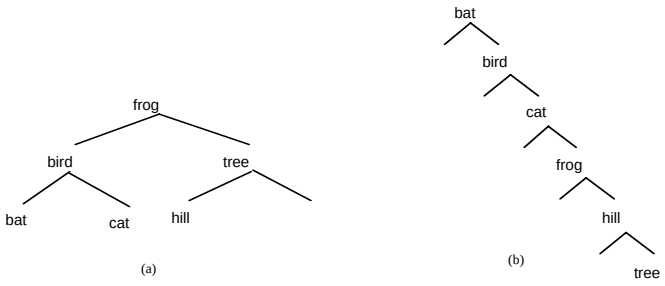


Figure 2.8 (a) A Balanced Binary Search Tree (b) A Binary Search Tree Which is Not Balanced

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) % HASHMAX` where `HASHMAX` is the 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, below, 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

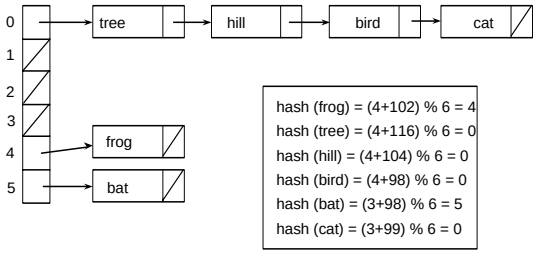


Figure 2.9 Hash Table Corresponding to the Words Entered for Figure 2.8(a)

depend on the sequence in which the words are encountered (though the sequence of words in a particular list could vary).

Exercises 2.3

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 in alphabetic order (worst case)
(c) 2^n-1 identifiers, perfectly balanced
(d) n identifers, and perfectly balanced

3. Write a program in C or C++ which will read a list of words (with no more than sixteen characters in a word) 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.

```
struct node {  struct node * left;
               char data[16];
               struct node * right;
            }  * 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 (see p. 51): `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 in Problem 6 to generate a collision for each identifier after the first.

2.4 Lex

The Unix programming environment includes several utility programs which are intended to improve the programmer's productivity. One such utility, called ***lex,*** is used to generate a lexical analyzer. The programmer specifies the words to be put out as tokens, using an extension of regular expressions. Lex then generates a C function, ***yylex()***, which, when compiled, will be the lexical analysis phase of a compiler.

Lex is designed to be used with another Unix utility called ***yacc***. Yacc is a parser generator which generates a C function, ***yyparse(),*** which contains calls to `yylex()` when it wants to read a token. However, it is also possible to use lex independently of yacc and yacc independently of lex. We will discuss yacc in greater detail in Section 5.3.

Lex may be useful for any software development project that requires lexical analysis, not just compilers. For example, a database query language that permits statements such as `Retrieve All Records For Salary >= $100,000` would require lexical analysis and could be implemented with lex.

2.4.1 Lex Source

The input to lex is stored in a file with a `.l` suffix (such as `fortran.l`). The structure of this file, consisting of three sections, is shown below:

```
C declarations and #includes
lex definitions
%%
lex patterns and actions
%%
C functions called by the above actions
```

The `"%%"` symbols are used as delimiters for the three sections of a lex program. The first and third sections are optional.

2.4.1.1 Section 1 of a Lex Program

The first section of the lex source file is for lex definitions and C declarations. The lex definitions are generally macro definitions which may be substituted in patterns in the second section. A macro definition consists of a name (preceded by no white space), followed by a lex pattern (see Section 2.4.1.2) to be substituted for that name. When the macro is used in a pattern, it must be enclosed in curly braces `{}`. For example,

```
MAC    ab*c
```

is a macro named `MAC`, and its value is `ab*c`. It could be used in a pattern:

```
hello{MAC}
```

which is equivalent to

```
helloab*c
```

Note that this is a simple and direct substitution! Be careful that you do not make assumptions about operator precedence in a pattern which uses macros.

The C declarations in the first section of the lex source file are those declarations which are to be global to the `yylex()` function. They should be inside lex curly braces. For example:

```
%{
#include "header.h"
#define  MAX  1000
char c;
%}
```

2.4.1.2 Section 2 of the Lex Program

The second section of the lex program, containing lex patterns and actions, is the most important part. This is sometimes called the ***rules*** section because these rules define the lexical tokens. Each line in this section consists of a pattern and an action. Each time the `yylex()` function is able to find input which matches one of the patterns, the associated action is executed. This pattern language is just an extension of regular expressions and is described below. In the following, `x` and `y` represent any pattern.

Pattern	Meaning
`c`	The char `"c"`
`"c"`	The char `"c"` even if it is a special char in this table
`\c`	Same as `"c"`, used to quote a single char
`[cd]`	The char `c` or the char `d`
`[a-z]`	Any single char in the range `a` through `z`
`[^c]`	Any char but `c`
`.`	Any char but newline
`^x`	The pattern `x` if it occurs at the beginning of a line
`x$`	The pattern `x` at the end of a line
`x?`	An optional x
`x*`	Zero or more occurrences of the pattern `x`
`x+`	One or more occurrences of the pattern `x`
`xy`	The pattern `x` concatenated with the pattern `y`
`x|y`	An `x` or a `y`
`(x)`	An `x`
`x/y`	An `x` only if followed by `y`

`<S>x`	The pattern `x` when lex is in start condition `S`
`{name}`	The value of a macro from definitions section
`x{m}`	`m` occurrences of the pattern `x`
`x{m,n}`	`m` through `n` occurrences of `x` (takes precedence over concatentation)

The ***start condition*** is used to specify left context for a pattern. For example, to match a number only when it follows a `$`:

```
             /* Enter start condition DOLLAR */
"$"                BEGIN DOLLAR;
             /* matches number preceded by $  */
<DOLLAR>[0-9]+     BEGIN 0;
             /* Return to default start condition */
```

In this example, the `BEGIN DOLLAR` statement puts lex into a start condition (`DOLLAR` is the name of the start condition). The pattern `<DOLLAR>[0-9]+` can be matched only when lex is in this start condition. The `BEGIN 0` statement returns lex to the original, default, start condition. Note that patterns written without any start condition specified can be matched regardless of the start condition state of lex. Lex must be informed of all start conditions in section 1 of the lex program with a `%start` declaration:

```
%start DOLLAR
```

This example should become more clear after the student has completed this section.

Right context can be specified by the `/` operation, in which `x/y` matches the pattern `x` when it occurs in right context `y`. For example the pattern `[a-z]*/[0-9]` matches the first three characters of the input string `abc3`, but does not match the first three characters of the string `abc@`.

The action associated with each pattern is simply a C statement (it could be a compound statement) to be executed when the corresponding pattern is matched. The second section of the lex source file will then consist of one rule on each line, each of which is a pattern followed by an action:

```
%%
pattern     action
pattern     action
pattern     action
.
.
.
%%
```

Each time `yylex()` is called it reads characters from ***stdin,*** the Unix standard input file (by default, `stdin` is pointing to the user's keyboard). It then attempts to match input characters with your patterns. If it cannot find a pattern which matches the

input beginning with the first character, it prints that character to ***stdout,*** the Unix standard output file (by default, `stdout` is pointing to the user's display) and moves on to the next character. We generally wish to write our patterns so that this does not happen; we do not want the input characters to be put out with the tokens. If there are several patterns which match the current input, `yylex()` chooses one of them according to these rules:

(1) The pattern that matches the longest possible string of characters starting at the current input is chosen.

(2) If there is more than one such pattern – all matching input strings of the same length – the pattern listed first in the lex source file is chosen.

After the pattern is chosen, the corresponding action is executed, and `yylex()` moves the current input pointer to the character following the matched input string. No pattern may match input on more than one line, though it is possible to have a pattern which matches a newline character - `\n`. An example of a lex source file is shown below:

```
%%
[a-z]+      printf ("alpha\n");           /* pattern 1 */
[0-9]+      printf ("numeric\n");         /* pattern 2 */
[a-z0-9]+   printf ("alphanumeric\n");    /* pattern 3 */
[ \t]+      printf ("white space\n");     /* pattern 4 */
.           printf ("special char\n");    /* pattern 5 */
\n          ;                             /* pattern 6 */
%%
```

The above lex source could be used to classify input strings as alphabetic, numeric, alphanumeric, or special, using white space as delimiters. Note that the action executed when a newline character is read is a null statement.

Lex declares global variables `char * yytext` and `int yyleng` for your use. `yytext` is a character string which always contains the input characters matched by the pattern, and `yyleng` is the length of that string. You may refer to these in your actions and C functions.

2.4.1.3 Section 3 of the Lex Program

Section 3 consists merely of C functions which are called by the actions in section 2. These functions are copied to the lex output file as is, with no changes. The user may also need to include a `main()` function in this section, depending on whether lex is being used in conjunction with *yacc*.

Sample Problem 2.4 (a)

Show the output if the input to `yylex()` generated by the lex program above is

```
abc123 abc 123?x
```

Solution:

```
alphanumeric
white space
alpha
white space
numeric
special char
alpha
```

The characters `abc` are matched by Pattern 1, and the characters `abc123` are matched by Pattern 3. Since `abc123` is longer than `abc`, Pattern 3 is selected. The input pointer is then advanced to the next character following `abc123`, which is a space. Pattern 4 matches the space. `yylex()` continues in that way until the entire input file has been read.

2.4.1.4 An Example of a Lex Source File

In this section we present an example of a lex source file. It converts numeric constants to internal floating-point form, and prints out a token called `Int` or `Float`, depending on the type of the number. It makes use of a standard C function called `sscanf()` which has the arguments:

```
sscanf (char * str, char * fmt, int * ptr_list)
```

This function is similar to the function `scanf()`, but differs in that the first argument is a string of characters. The function reads from this string of characters, rather than from the standard input file, `stdin`. In our example it reads from the string `yytext`, which always contains the characters matched by the lex pattern. The other arguments to `sscanf()` are a format string and one or more pointers to allocated storage. As with `scanf()`, the characters are converted to another form, as specified by the format string, and stored in the allocated storage areas pointed to by the pointers. The use of `sscanf()` eliminates the need to convert numeric constants to internal formats as we did in Sample Problem 2.2. The example of a lex source file is shown below:

```
INT       [0-9]+
EXP       ([eE][+-]?{INT})
%{
```

```
int i;
float f;
%}
%%
{INT}                 {sscanf (yytext, "%d", &i);
                       cout << "Int" << endl;}
{INT}\.{INT}?{EXP}?   {sscanf (yytext, "%lf",&f);
                       cout << "Float" << endl;}
.                     ;  /* gobble up anything else */
%%
main ()
{ yylex(); }
yywrap ()
{ }
```

Note that the parentheses around the definition of `EXP` in this example are necessary because this macro is called below and followed by a `'?'` in the pattern which matches real numbers (numbers containing decimal points). A macro call is implemented with a simple substitution of the macro definition. Consequently, if the macro definition had not been in parentheses, the substitution would have produced:

```
{INT}\.{INT}?[eE][+-]?{INT}?
```

in which the last `'?'` is operating only on the last `{INT}` and not on the entire exponent.

Sample Problem 2.4 (b)

Improve the lex program shown above to recognize identifiers, assignment operator (=), arithmetic operators, comparison operators, and the keywords `if, while, else` for a language such as C++.

Solution:

```
INT      [0-9]+
EXP      ([eE][+-]?{INT})
IDENT [a-zA-Z][a-zA-Z0-9]*
%{
int i;
float f;
%}
```

```
%%
if                          printf ("keyword - if\n");
while                       printf ("keyword - while\n");
else                        printf ("keyword - else\n");
{IDENT}                     printf ("identifier\n");
\+|\-|\*|\/                 printf ("arithmetic operator\n");
"="                         printf ("assignment\n");
"=="|\<|\>|"<="|">="|"!="   printf ("comparison operator\n");
{INT}                        {sscanf (yytext, "%d", &i);
                              printf ("Int\n");}
{INT}\.{INT}?{EXP}?          {sscanf (yytext, "%lf",&f);
                              printf ("Float\n");}
.                            ;  /* gobble up anything else */
%%
main ()
{ yylex(); }
yywrap()
{ }
```

Note that the keywords are all reserved words – i.e., they are reserved by the compiler for a specific purpose and may not be used as identifiers. That is why they must precede the pattern for identifiers in the lex source file.

2.4.2 Running Lex

There is now a two-step process to generate your program:
(1) Use lex to generate the C function. It is placed in a file called `lex.yy.c`
(2) Use the C compiler to compile that file. If you have used C++ constructs which are not C compatible in your actions or functions, you must compile with a C++ compiler.
It may be necessary to link the ***lex library*** with the output of the C compiler. This is done with the `-ll` option, but for our examples it won't be necessary.

Assuming your lex source file is `language.l`, these two steps are shown below:

```
$ lex language.l
$ cc lex.yy.c -o language
```

Some versions of lex may include a call to a function named `yywrap()`. This call always takes place when the entire input file has been scanned and `yylex()` is ready to terminate. The purpose of `yywrap()` is to give you the opportunity to do additional processing in order to "wrap" things up before terminating. In most cases you can make this a null function if necessary and include it in section 3 of your lex source file.

In Chapter 5 we will use the `yylex()` function as a subprogram rather than as a main program. The `yylex()` function will be called repeatedly, scan the input file, and return an integer code each time it finds a complete token. In general it will not print to `stdout` unless it detects a lexical error.

Exercises 2.4

1. Write a lex "program" to print out the following token class numbers for C++ source input:

(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
(7) Keyword (`if else while do for const`)
Comments /* using this method */
// or this method, but don't print a token number

2. Show the output of the following lex program for each of the given input strings:

```
CHAR       [a-z][0-9]
%%
{CHAR}*    printf ("pattern 1\n");
{CHAR}x    printf ("pattern 2\n");
({CHAR})*  printf ("pattern 3\n");
.          printf ("pattern 4\n");
\n         ;
%%
```

(a) `a1b2c3`
(b) `abc3+a123`
(c) `a4x+ab22+a22`

2.5 Case Study: Lexical Analysis for MiniC

In this section we present a description of the lexical analysis phase for the subset of C++ we call ***MiniC***. This represents the first phase in our case study – a complete MiniC compiler. The lexical analysis phase is implemented with the lex utility, and the lex source file is shown in its entirety in Appendix B.2 (refer to the files `MiniC.l` and `MiniC.h`).

The MiniC case study is implemented as a two-pass compiler. The syntax and lexical phases combine for the first pass, producing a file of atoms, and the code generator forms the second pass. This means that control is initially given to the syntax phase, which calls `yylex()` each time it needs a token. Consequently, our lex program (which is used to generate the `yylex()` function) returns a token class to the calling program in the action associated with each pattern.

The lex source file is divided into three sections, separated by the `%%` symbol as described above in Section 2.4. We now describe the content of each of these sections in `MiniC.l`.

The first three lines, shown below, are lex macro definitions:

```
INT    [0-9]+
EXP    ([eE][+-]?{INT})
NUM    {INT}\.?{INT}?{EXP}?
```

An `INT` is defined to be a string of one or more digits; an `EXP` is the exponent part of a number (the sign is optional), and a `NUM` is any numeric constant (in MiniC we constrain numeric constants to begin with a digit, but the decimal point, fractional part, and exponent part are all independently optional).

Also in the first section, the macro definitions are followed by global C declarations to be copied verbatim into the output file. The `stdlib.h` header file allows for ANSI C compatibility, and the three function declarations specify that the functions return a type `ADDRESS`, representing a run-time address on the target machine. The type definition for `ADDRESS` is in the header file `MiniC.h`, and is explained further in Chapter 6.

The second section of the lex source file contains the patterns and actions which define the tokens to be returned to the calling program. Single character tokens, such as parentheses, semicolon, arithmetic operations, etc., can be returned as the token class (the ascii code of the character is the token class). Other token classes are defined in the yacc file, `MiniC.y`, with `%token` declarations. They are assigned integer values over 255 by yacc, but we need not be concerned with those values.

The MiniC keywords each constitute a distinct token class, e.g. the keyword `for` is returned as a `FOR` token, the keyword `while` is returned as a `WHILE` token, etc. Since MiniC is case sensitive, all keywords must be lower case. In languages which are not case sensitive, such as Pascal, we would have to accept keywords with mixed case, such as `wHiLE`. This could be done with a pattern like `[wW][hH][iI][lL][eE]` for the `WHILE` token. Alternatively, we could use a filter on the source program to convert

all upper-case letters (excepting those in string constants) to lower case before proceessing by the compiler.

The six keywords are found in the MiniC definition in Appendix A, where they are the symbols beginning with lower-case letters. Each keyword forms its own token class, with no value part, except for the type declarations `int` and `float`, which have value parts 2 and 3, respectively. These values are stored in the global variable `yylval.code` (the `.code` qualifier will be explained in the case study for Chapter 5), and are used to indicate the type of an identifier in the symbol table.

The keywords are followed by the six comparison operators, forming the token class `COMPARISON`. Again, the value part, stored in `yylval.code`, indicates which of the six comparison operators has been matched.

The pattern for identifiers is `[a-zA-Z][a-zA-Z0-9_]*`, which indicates that an identifier is a string of letters, numeric digits, and underscores which begins with a letter (either lower or upper case). In addition to returning the token class `IDENTIFIER`, the action also calls the `searchIdent()` function, which installs the identifier in the symbol table (implemented as a hash table) and returns a run-time address for the identifier.

The pattern for numeric constants is simply the macro `{NUM}` as defined in the first section. In this case the action calls the function `searchNums()` to install the constant in a table of numeric constants (implemented as a binary search tree, just to expose a different technique to the student).

White space (spaces, tabs, and newline characters) may serve as delimiters for tokens, but white space does not, itself, constitute a token, since MiniC, like C++, is free format. Both kinds of comments, C style and C++ style , are recognized. C style comments are enclosed in /* and */, while C++ style comments begin with // and continue to the end of the line. Start conditions are used to recognize the fact that the scanner is inside a comment, and that no tokens are to be returned until the end of the comment is found. When processing comments, it is important that we not use a pattern such as `"/*".*"*/"`. This would be incorrect since

```
/* comment1 */  x = 2;   /* comment2 */
```

would be taken as one long comment.

The final pattern, a single period, matches any single character not matched by any of the preceding patterns. This character is itself returned as a token class (its ascii code is the class). For characters such as parentheses, semicolons, arithmetic operations, etc., this is fine. Other, unexpected characters, such as `$%#` etc., will also be returned as tokens and ultimately will cause the syntax phase to generate a syntax error message.

The third section of the lex source file (after the second `%%` delimiter) contains supporting functions called either from the actions in the second section or from the main program. The first such supporting function, `yywrap()`, is called from the main program when the entire MiniC program has been read. It's purpose is to permit housekeeping such as releasing unneeded storage, closing files, etc. In our case, we do not need to do any of these, so we just return with a successful return code.

The next supporting function, `searchIdent()`, is called from the action for `IDENTIFIER` tokens and is used to install the identifier in the symbol table. The symbol table is implemented as a hash table of `HashMax` linked lists, as described in Section 2.3.3. The hash function simply sums the characters in the identifier (stored in `yytext`) and returns a subscript in the range `0` to `HashMax-1`. The function then searches the appropriate linked list. If the identifier is not found, it installs the identifier in the hash table by allocating space (with the standard `malloc()` function) for a node and the identifier, as shown in Figure 2.10. The function returns the run-time address of the identifier, provided by the `alloc()` function. The global variable `dcl` is `TRUE` if the compiler is scanning a declaration. In this case we expect identifiers not to be in the symbol table. If `dcl` is `FALSE`, the compiler is not scanning a declaration, and we expect identifiers to be in the symbol table, since they should have been declared previously. This enables us to detect undeclared, and multiply declared, identifiers.

In Figure 2.10, below, we show a diagram of what the symbol table would look like after two identifiers, `b` and `cad`, have been scanned. Each symbol table entry consists of four parts: 1) a pointer to the identifier, 2) a pointer to the next entry in the linked list, 3) a code for the type of the identifier, and 4) a run-time address for the value of this identifier. The run-time value of `b` will be stored at location 0 and the run-time value of `cad` will be stored at location 1.

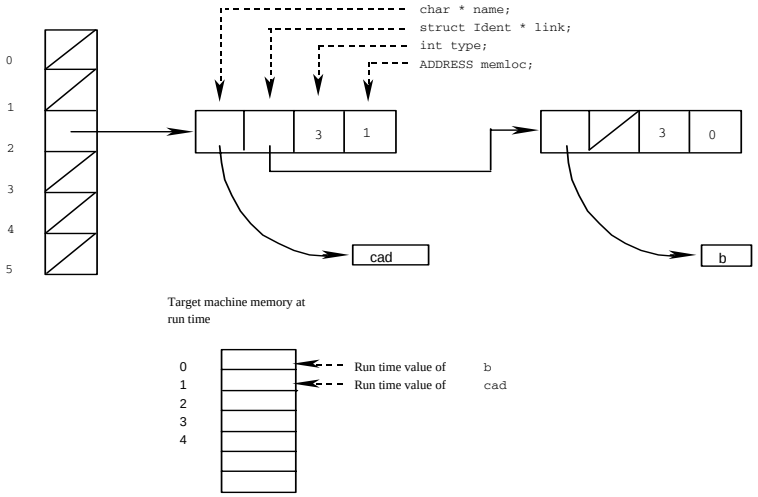


Figure 2.10 A Hash Table Storing the Identifiers `cad` and `b`, with Target Machine Memory

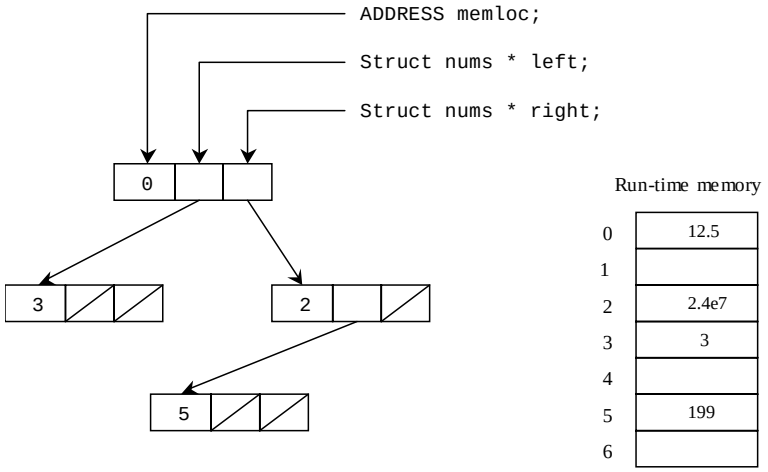


Figure 2.11 Binary Search Tree Storing the Constants 12.5, 2.4e7, 3, and 199, With Target Machine Memory

The `searchNums()` supporting function is very similar to `search-Ident()`. It installs numeric constants in a binary search tree, in which each constant found in the MiniC program is stored once, as shown, above, in Figure 2.11. A pointer to the root node of the tree is in the variable `numsBST`. The constant, stored as characters in the string, `yytext`, must be converted to a numeric data format. This is done with the `sscanf()` function, which is similar to `scanf()`, but reads from a string – `yytext` in this case – rather than from the `stdin` file. The `searchNums()` function also returns the run-time address of the numeric constant, and the constant is stored in an array, `memory`, which will become part of the output file when code is generated. This will be described in the case study section for Chapter 6.

Exercises 2.5

1. Use lex to write a *filter* which converts all alphabetic characters to lower case, unless they are inside double quote marks. The quote mark may be included in a quoted string by using the backslash to quote a single character. Backslashes also may be included in a string. Assume that quoted strings may not contain newline characters. Your filter should read from `stdin` and write to `stdout`. Some examples are:

Input	Output
`"A String\"s Life" Is Good`	`"A String\"s Life" is good`
`OPEN "C:\\dir\\File.Ext"`	`open "C:\\dir\\File.Ext"`

2. Revise the lex source file `MiniC.l` shown in Appendix B.2 to permit a `switch` statement and a `do while` statement in MiniC:

```
SwitchStmt  →  switch (Expr) { CaseList }
CaseList    →  case NUM : StmtList
CaseList    →  case default: StmtList
CaseList    →  case NUM : StmtList  CaseList
Stmt        → break ;

DoStmt  →  do Stmt while ( Expr )
```

You will not be able to run lex on `MiniC.l` since it is designed to be used with the yacc utility, which we will discuss in Chapter 5.

3. Revise the *macro definitions* in the lex source file `MiniC.l` shown in Appendix B.2 to exclude numeric constants which do not begin with a digit, such as `.25` and `.03e-4`. You will not be able to run lex on `MiniC.l` since it is designed to be used with the yacc utility, which we will discuss in chapter 5.

4. Rather than having a separate token class for each MiniC keyword, the scanner could have a single class for all keywords, and the value part could indicate which keyword has been scanned (e.g. `int` = 1, `float` = 2, `for` = 3, ...). Show the changes needed in the file `MiniC.l` to do this.

2.6 Chapter Summary

Chapter 2, on ***Lexical Analysis***, began with some introductory theory of formal languages and automata. A *language*, defined as 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 lexical scanner generator, lex, which makes use of regular expressions in specifying patterns to match lexical tokens in the source language. The lex source file consists of three sections: (1) lex macros and C declarations; (2) rules, each rule consisting of a pattern and action; and (3) supporting functions. We concluded the chapter with a look at a lex program which implements the lexical scanner for our case study – MiniC.

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 C++ 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 as a means of not only 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***.

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 $\alpha \rightarrow \beta$, where α and β are arbitrary strings of terminals and nonterminals, and α 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:

$$S \Rightarrow \alpha \Rightarrow \beta \Rightarrow \gamma \Rightarrow ... \Rightarrow x$$

where α, β, γ 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 → 0S0
2. S → 1S1
3. S → 0
4. S → 1

An example of a derivation using this grammar is:

S ⇒ 0S0 ⇒ 00S00 ⇒ 001S100 ⇒ 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 → ASB
2. S → ε
3. A → a
4. B → b

S ⇒ ASB ⇒ AASBB ⇒ AaSBB ⇒ AaBB ⇒ AaBb ⇒ Aabb ⇒ 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) = { ε, 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 (a)

Show three different derivations using the grammar shown below:

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

Solution

```
S  ⇒  a S A  ⇒  a B A A  ⇒  a B a b A  ⇒  a B a b a b
      ⇒  a b A a b a b  ⇒  a b a b a b a b
S ⇒ a S A ⇒ a S a b  ⇒  a B A a b  ⇒  a b A A a b
      ⇒  a b a b A a b  ⇒  a b a b a b a b
S  ⇒ B A  ⇒  b A A  ⇒  b a b A  ⇒  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: ababababab.

3.0.2 Classes of Grammars

In 1959 Noam Chomsky, a linguist, suggested a way of classifying grammars according to complexity. 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 ε is permitted on the right side of the arrow only). An example of an unrestricted rule would be:

SaB → 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 α,β and γ are 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 → 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  →  aABb
```

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

A → aB
or
A → 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.

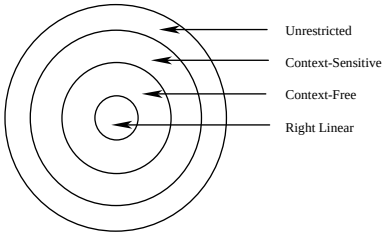


Figure 3.1 Classes of Grammars

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, above, 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.

G3:

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

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

The student should perform other derivations to understand that

$L(G3) = \{\varepsilon, abc, aabbcc, aaabbbccc, \ldots\}$
$= \{a^n b^n c^n\}$ 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 (b):

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):

Solution:

1.	`aSb → aAcBb`	Type 1, Context-Sensitive
2.	`B → aA`	Type 3, Right Linear
3.	`a → ABC`	Type 0, Unrestricted
4.	`S → aBc`	Type 2, Context-Free
5.	`Ab → b`	Type 1, Context-Sensitive
6.	`AB → BA`	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 → a S b
```

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

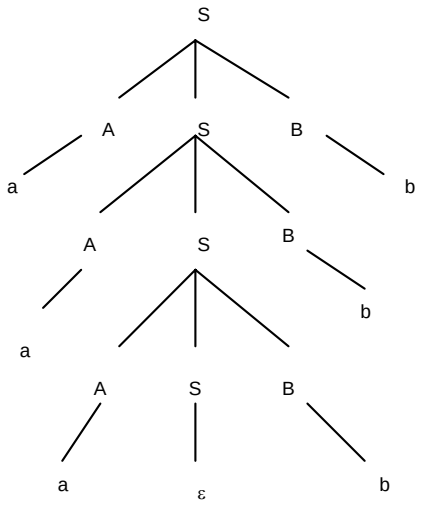


Figure 3.2 A Derivation Tree for `aaabbb` Using Grammar G2

```
<S> ::= a <S> b | ε
```

which is the BNF version of two grammar rules:

```
S → a S b
S → ε
```

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 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 )
```

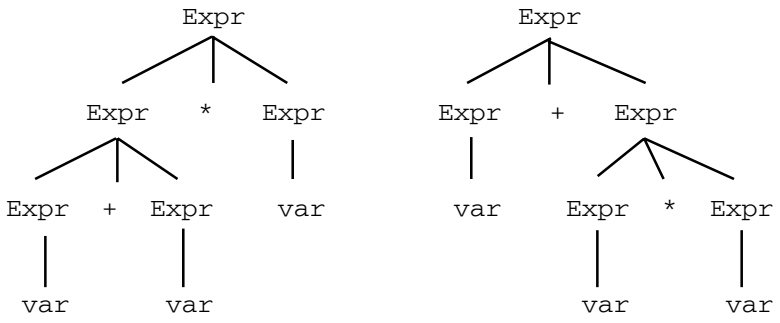


Figure 3.3 Two Different Derivation Trees for the String var + var * var

```
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 Section 3.1.

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.

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:

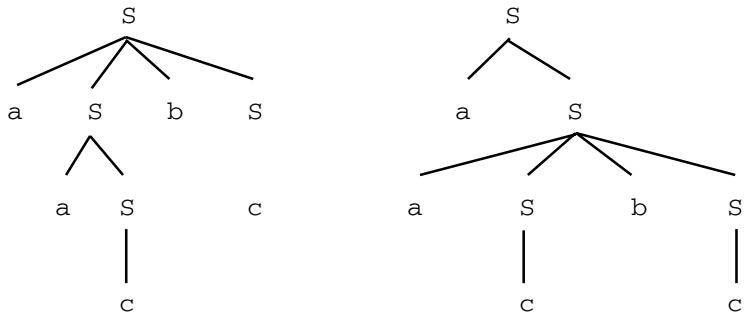
1. A finite set of states, one of which is designated the ***starting state***.

Sample Problem 3.0 (c)

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 → a S b S
2. S → a S
3. S → c

Solution:



S ⇒ a S b S ⇒ a a S b S ⇒ a a c b S ⇒ a a c b c
S ⇒ a S ⇒ a a S b S ⇒ a a c b S ⇒ a a c b c

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

2. A finite set of input symbols, the ***input alphabet***.

3. 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.

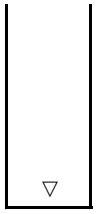
4. A state transition function which takes as arguments the current state, the current input symbol, and the symbol currently on top of the stack; its result is the new state of the machine.

5. 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).

6. 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.

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

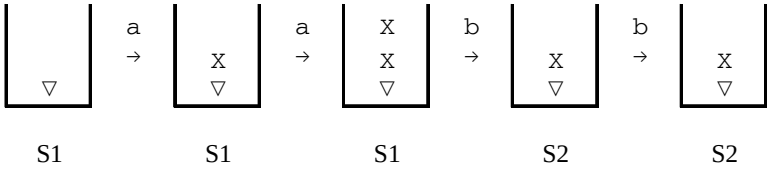


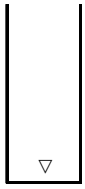
Figure 3.5 Sequence of Stacks as Pushdown Machine of Figure 3.4 Accepts the Input String aabb

7. 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, above, in Figure 3.4, in which the rows are labeled by stack symbols and the columns are labeled by input symbols. The ↩ character is used as an ***endmarker***, indicating the end of the input string, and the ▽ 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

S1	(	)	↩
X	Push (X) Advance S1	Pop Advance S1	Reject
▽	Push (X) Advance S1	Reject	Accept



Initial Stack

Figure 3.6 Pushdown Machine to Accept Any String of Well-Balanced Parentheses

from the machine. The language of strings accepted by this machine is $a^n b^n$ where $n \bullet 0$ – 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 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 (see p. 80). (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.

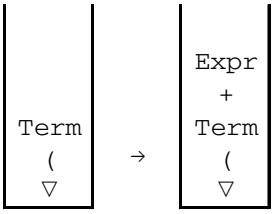


Figure 3.7 Effect on Stack of `Rep (Term, +, Expr)`

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 included 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 (see p. 81) 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 `Ep` and `Lp` 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 machine, we can write a right linear grammar which specifies the same language accepted by the finite state machine.

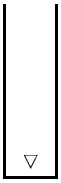
There are algorithms which can be used to produce any of these three forms (finite state machines, right linear grammars, and regular expressions), given one of the other two (see, for example, Hopcroft and Ullman [1979]). However, here we rely on the student's ingenuity to solve these problems.

We have a similar correspondence between machines and context-free languages. Any language which can be accepted by a deterministic pushdown machine can

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

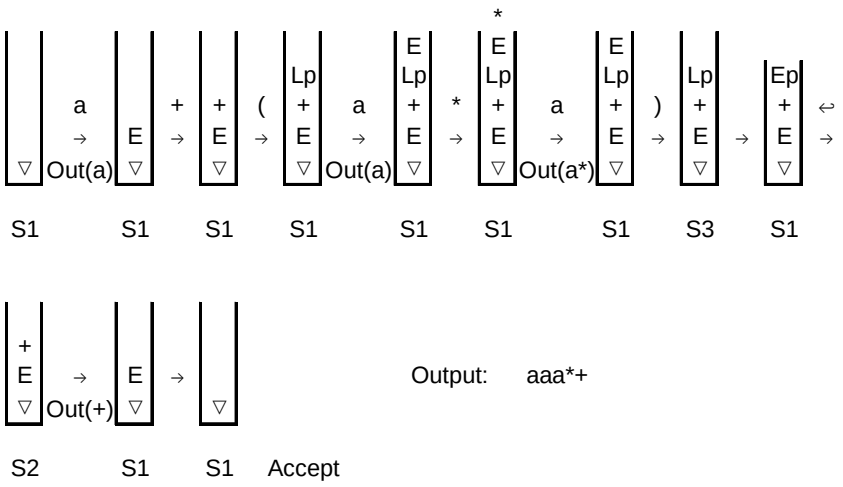
Figure 3.8 Pushdown Translator for Infix to Postfix Expressions

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, P_c, of palindromes over the alphabet {0,1} with centermarker, c. $P_c = wcw^r$, where w is any string of 0's and 1's, and w^r is w reversed. A grammar for P_c is shown below:

Sample Problem 3.0 (d)

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:



Output: aaa*+

Sample Problem3.0 (e)

Give a right linear grammar for each of the languages specified in Sample Problem 2.0 (a) (see p. 33).

Solution:

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

1. `S → 0`
2. `S → 1S`
3. `S → 0A`
4. `A → 1`
5. `A → 1A`
6. `A → 0S`

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

1. `S → 1S`
2. `S → 0S`
3. `S → 1A`
4. `A → 1B`
5. `B → 1C`
6. `B → 1`
7. `C → 1C`
8. `C → 0C`
9. `C → 1`
10. `C → 0`

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

1. S → 1S
2. S → 0A
3. A → 1A
4. A → 0B
5. B → 1B
6. B → 0C
7. B → 0
8. C → 1C
9. C → 1

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

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

S → 0S0
S → 1S1
S → 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 P_c. 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:

S → 0S0
S → 1S1
S → 0
S → 1
S → ε

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

Exercises 3.0

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

(a)
```
S → a S
S → b A
A → b S
A → c
```

(b)
```
S → a B c
B → A B
A → B A
A → a
B → ε
```

(c)
```
S → a S B c
a S A → a S b b
B c → A c
S b → b
A → a
```

(d)
```
S → a b
a → a A b B
A b B → ε
```

2. Classify the grammars of Problem 1 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 Problem 5.

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 → S

(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 >= 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 the binary representation of the integer i, and $(N_i)^r$ is N_i written right to left (reversed). Example for, i=19:
10011c00101
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.

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↩` (b) `(a+a)*a↩`
(c) `(a)↩` (d) `((a))↩`

10. Show a grammar and 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 centermarker `c`. This is the language, P_c, referred to in Section 3.0.5.

12. Show a grammar for the language of valid regular expressions over the alphabet `{0,1}`. Hint: Think about grammars for arithmetic expressions.

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 (see p. 75) 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, below, 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:

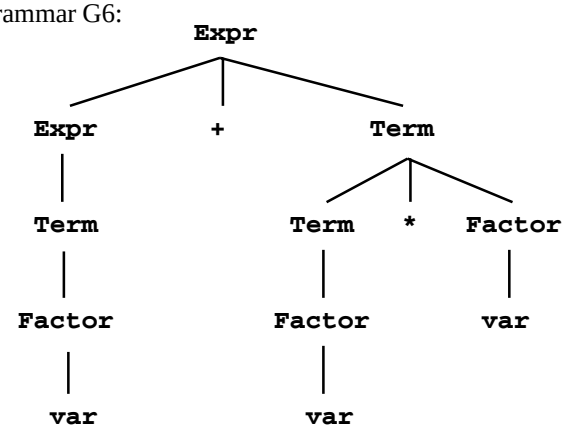


Figure 3.9 A Derivation Tree for `var + var * var` Using Grammar G5

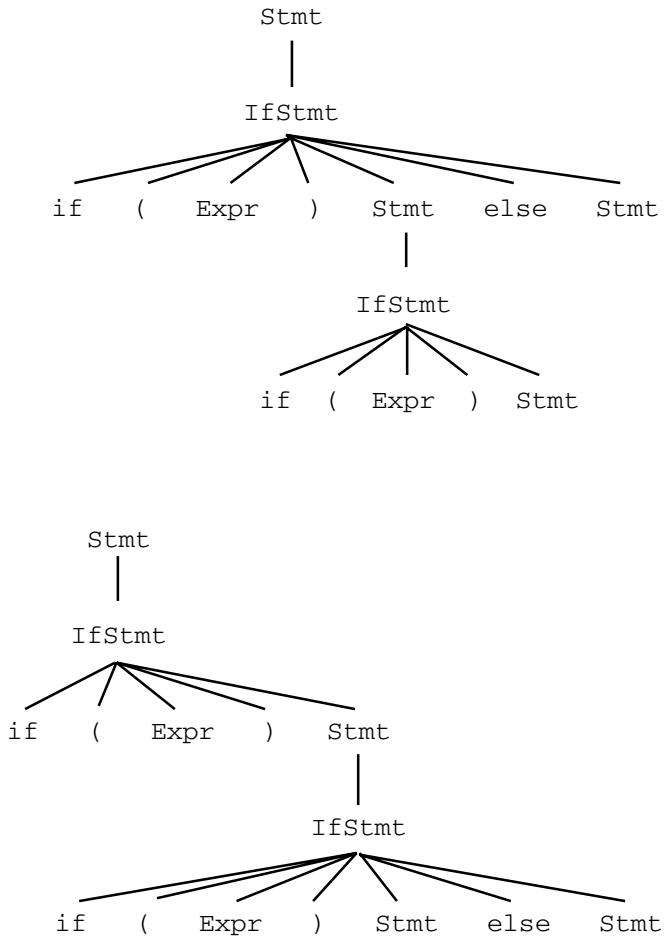


Figure 3.10 Two Different Derivation Trees for: `if ( Expr ) if ( Expr ) Stmt else Stmt`

G6:

```
1.    Stmt → IfStmt
2.    IfStmt → if ( Expr ) Stmt
3.    IfStmt → if ( Expr ) Stmt else Stmt
```

Think of grammar G6 as part of a larger grammar in which the nonterminal `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 (Expr) if (Expr) Stmt else Stmt` are shown, above, in Figure 3.10. In this grammar, an `Expr` is interpreted as False (0) or True (non-zero), and a `Stmt` is

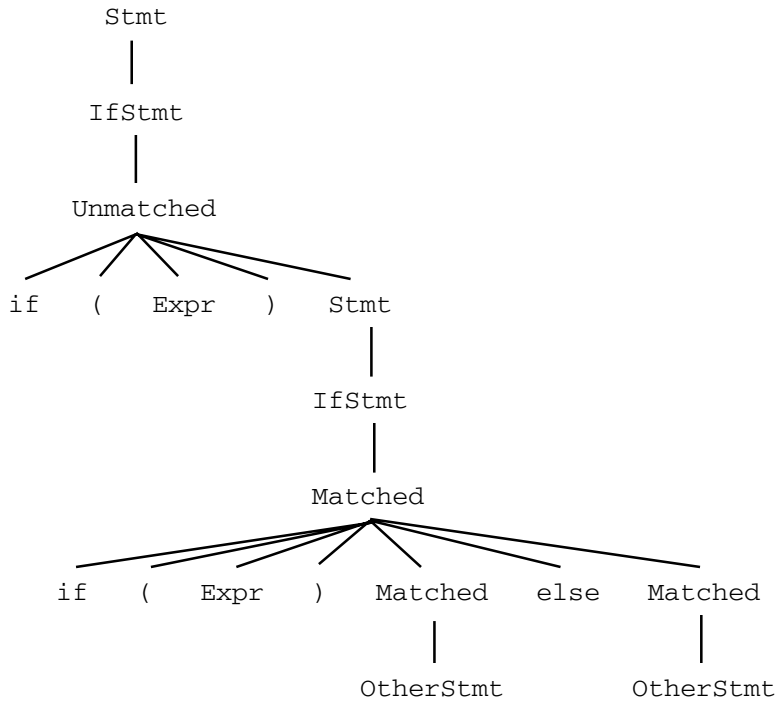


Figure 3.11 A Derivation Tree for `if ( Expr ) if ( Expr ) OtherStmt else OtherStmt` using Grammar G7

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 ( Expr ) Matched else Matched
5.    Matched → OtherStmt
6.    Unmatched → if ( Expr ) Stmt
7.    Unmatched → if ( Expr ) Matched else Unmatched
```

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, expression, for`, ...). A derivation tree for the string `if ( Expr ) if ( Expr ) OtherStmt else OtherStmt` is shown in Figure 3.11 (see p. 89).

Exercises 3.1

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 Problem 2 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 → a A b          S → x B y
A → b A a          B → y B x
A → a              B → x
```

Which grammar in Section 3.1 is *isomorphic* to the grammar of Problem 4 in Section 3.0?

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

(a) `if ( Expr ) OtherStmt`
(b) `if ( Expr ) OtherStmt else if ( Expr ) OtherStmt`
(c) `if ( Expr ) if ( Expr ) OtherStmt else Stmt else OtherStmt`
(d) `if ( Expr ) if ( Expr ) if ( Expr ) Stmt else OtherStmt`

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, 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, below, 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.

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 and Ullman [1972], for example).

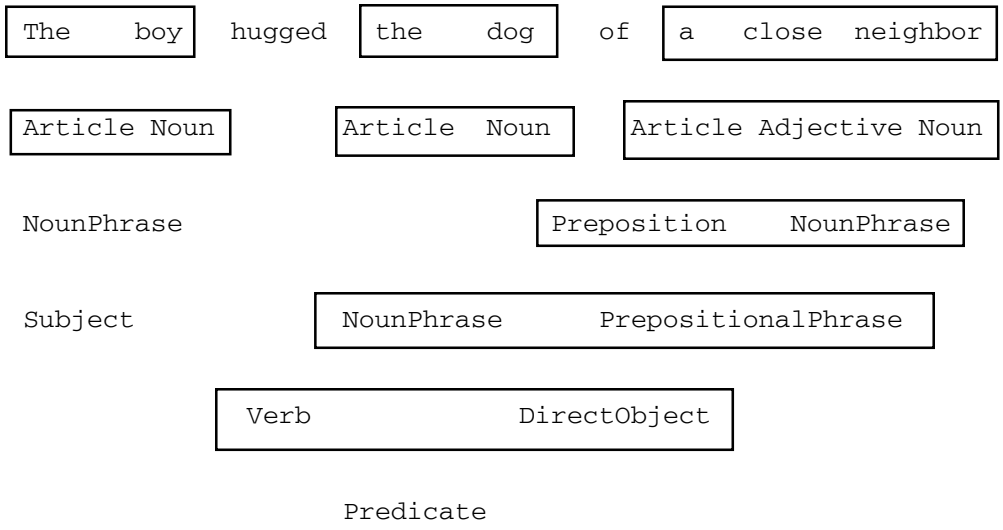


Figure 3.12 Diagram of an English sentence

3.3 Chapter Summary

Chapter 3, 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 (see p. 92) 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:

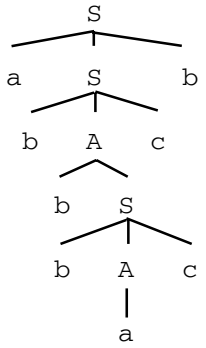


Figure 4.1 A Derivation Tree for `abbbaccb`

G8:

```
1.    S  →  a S b
2.    S  →  b A c
3.    A  →  b S
4.    A  →  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 (see p. 95) 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

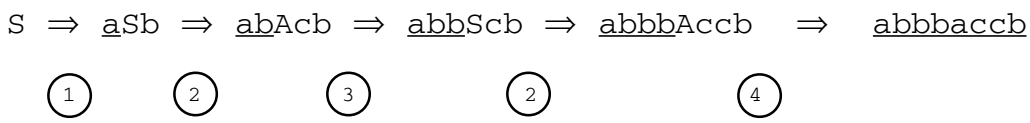


Figure 4.2 Sequence of Events in a Top Down Parse

symbols are the ones which have been read by the parser. Careful study of Figures 4.1 (see p. 94) and 4.2, above, 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 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*`; 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).

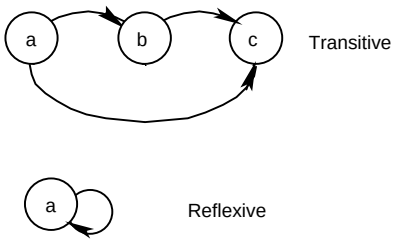


Figure 4.3 Reflexive and Transitive Elements of a Relation

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, above, 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

Show R1* the reflexive transitive closure of 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 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.

Exercises 4.0

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

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

(b)
```
(a,a)
(a,b)
(b,b)
```

(c)
```
(a,b)
(c,d)
(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 Pascal 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 → aα

(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:

S → aSb
S → b

G10:

S → aSb
S → ε

G11:

S → aSb
S → 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 (see p. 99) 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

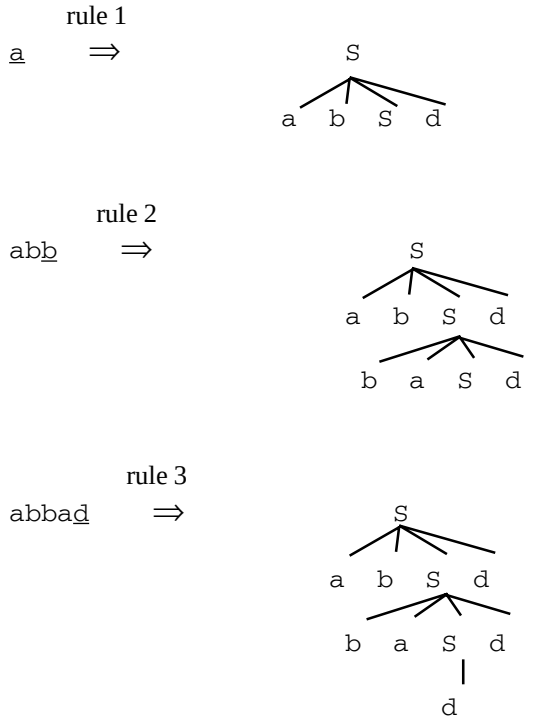


Figure 4.4 Using the Input Symbol to Guide the Parsing of the String abbaddd

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.

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 → aSB
2. S → b
3. B → a
4. B → 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 (▽) 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 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 ▽) in order for the string to be accepted. The extended pushdown machine for grammar G13 is shown in Figure 4.5, below. The sequence of stack configurations produced by this machine for the input aba is shown in Figure 4.6 (see p. 101).

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 endmarker ↩) and each row labeled by a nonterminal or terminal symbol (and bottom marker ▽).

	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

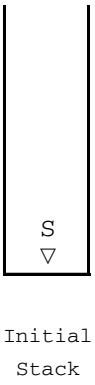


Figure 4.5 Pushdown Machine for Grammar G13

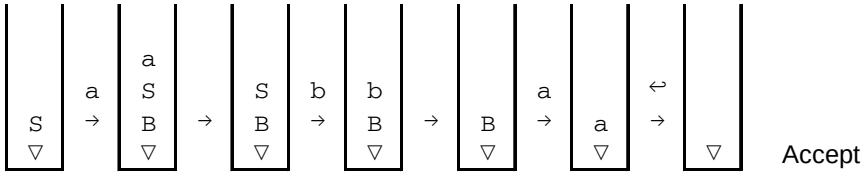


Figure 4.6 Sequence of Stacks for Machine of Figure 4.5 for Input `aba`

2. For each grammar rule of the form `A` → `a`α, fill in the cell in row `A` and column `a` with: `REP(`α^r`a), retain`, where α^r represents α 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 ▽ and column ↩ with `Accept`.

5. Fill in all other cells with `Reject`.

6. Initialize the stack with ▽ 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 *yacc* in the section on bottom up parsing.

4.1.2 Recursive Descent Parsers for Simple Grammars

A second method for implementing a parser for simple grammars is to use a method known as ***recursive descent***. In this method, the parser is written using a procedure oriented language, such as Pascal or C. A function is written for each nonterminal in the grammar. The purpose of this function 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 function then handles each succeeding symbol in the rule; it handles nonterminals by calling the corresponding functions, and it handles terminals by reading another input symbol. For example, a recursive descent parser for grammar G13 is shown on the next page:

```
char  inp;
```

```
void parse ()
   { cin >> inp;
     S ();
     if (inp=='⇐') accept();
     else reject();
   }

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

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

Note that the main function (`parse`) reads the first input character before calling the function for nonterminal `S` (the starting nonterminal). Each function assumes that the initial input symbol in the example it is looking for has been read before that function has been called. It then uses the selection set to determine which of the grammar rules defining that nonterminal should be applied. The function `S` calls itself (because the nonterminal `S` is defined in terms of itself), hence the name *recursive descent*. When control is returned to the main program, it must ensure that the entire input string has been read before accepting it. The functions `accept()` and `reject()`, which have been omitted from the above program, simply indicate whether or not the input string is in the language of the grammar. The student should perform careful hand simulations of this program for various input strings, to understand it fully.

Exercises 4.1

Sample Problem 4.1

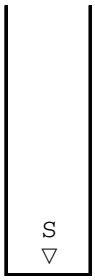
Show a one state pushdown machine and a recursive descent parser (show functions `S()` and `A()` only) for the following grammar:

1. S → 0 S 1 A
2. S → 1 0 A
3. A → 0 S 0
4. A → 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 pushdown machine is shown below:

	0	1	↩
S	Rep (A1S0) Retain	Rep (A01) Retain	Reject
A	Rep (0S0) Retain	Rep (1) Retain	Reject
0	pop Advance	Reject	Reject
1	Reject	pop Advance	Reject
▽	Reject	Reject	Accept



Initial Stack

The recursive descent parser is shown below:

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

```
            if (inp=='0') cin >> inp;
            else reject();
            A ();
         }                                  // end rule 2
      else reject();
   }

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

1. Determine which of the following grammars are s*imple*. 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 (see p. 100) 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 ε 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:

A → ε

(where A 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 → a A S
2.    S → b
3.    A → c A S
4.    A → ε
```

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 ↩) which can immediately follow an A in an intermediate form derived from S↩, where S is the starting nonterminal. For grammar G14, above, the follow set of S is {a,b,↩} and the follow set of A is {a,b}, as shown by the following derivations:

S↩ ⟹ aAS↩ ⟹ acASS↩ ⟹ acASaAS↩
⟹ acASb↩ Fol(S) = {a,b,↩}

S↩ ⟹ aAS↩ ⟹ aAaAS↩
⟹ aAb↩ 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

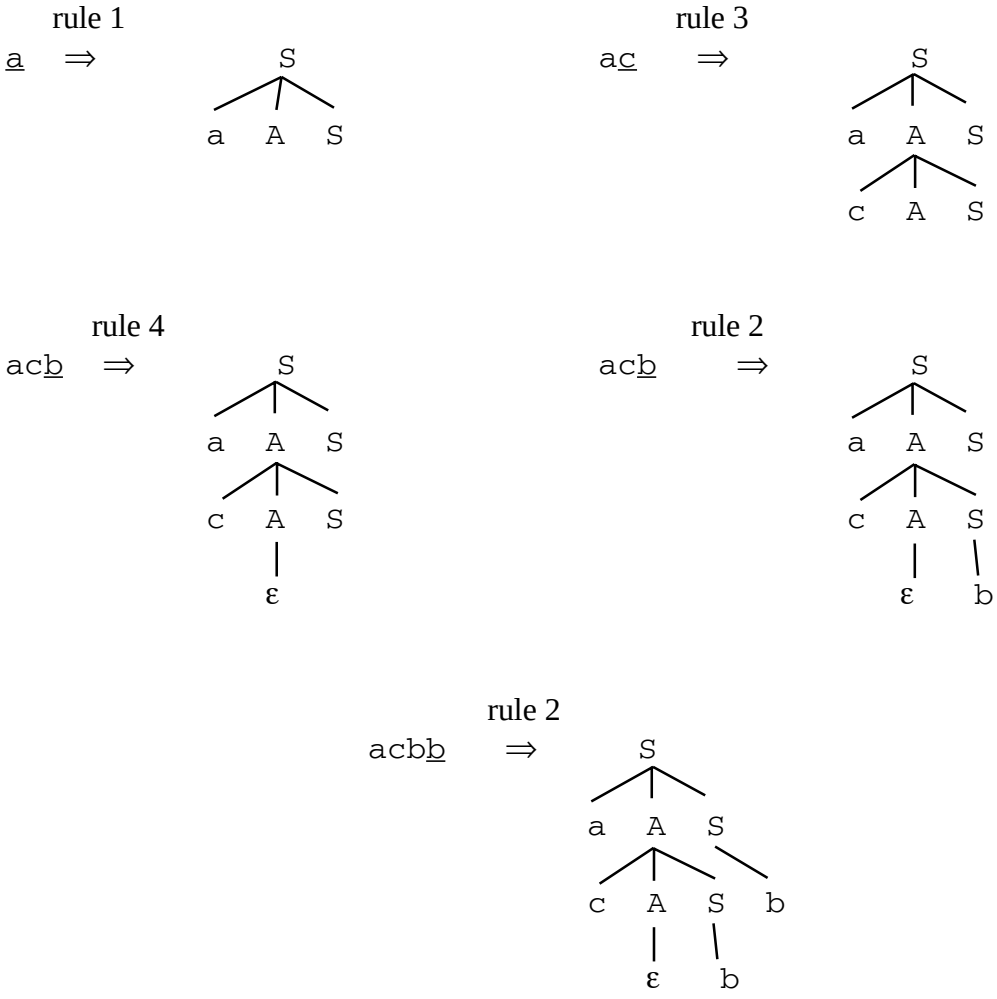


Figure 4.7 Construction of a Parse Tree for acbb Using Selection Sets

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.

4.2.1 Pushdown Machines for Quasi-Simple Grammars

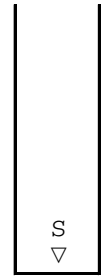
To build a pushdown machine for a quasi-simple grammar, we need to add only one step to the algorithm given in Section 4.1.1 (see p. 99). We need to apply an ε rule by simply popping the nonterminal off the stack and retaining the input pointer. We do this only when the input symbol is in the follow 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, above. Note, in particular, that the entries in columns `a` and `b` for row `A (Pop, Retain)` correspond to the ε rule (rule 4).

	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	Accept



Initial Stack

Figure 4.8 A Pushdown Machine for Grammar G14

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 function 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 ()
   {  cin >> inp;
      S ();
      if (inp=='↩') accept();
      else reject();
   }
```

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

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

Note that rule 4 is applied in function `A()` when the input symbol is `a` or `b`. Rule 4 is applied by returning to the calling function without reading any input characters. This is done by making use of the fact that C++ 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 (see p. 108), for the row labeled `A`, we have filled in `Pop, Retain` under the input symbols `a` and `b`, but `Reject` under the input symbol ↔; 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

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 functions `S()` and `A()` only) corresponding to this grammar.

1. S → b A b
2. S → a
3. A → ε
4. A → 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 ↩) which could follow an A in a derivation from S↩.

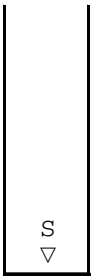
$S\hookleftarrow \Longrightarrow b\underline{Ab}\hookleftarrow$

We cannot find any other terminals that can follow an A in a derivation from S↩. Therefore, FOL(A) = {b}. The selection sets can now be listed:

```
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 disjoint 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
         {  cin >> inp;
            A ();
            if (inp=='b') cin >> inp;
            else reject();
         }                             // end rule 1
      else if (inp=='a') cin >> inp;   // apply rule 2
           else reject();
   }

void A ()
   {  if (inp=='b') ;                  // apply rule 3
      else if (inp=='a')               // apply rule 4
         {  cin >> inp;
            S ();
            if (inp=='a') cin >> inp;
            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.

Exercises 4.2

1. Show the *sequence of stacks* for the pushdown machine of Figure 4.8 (see p. 108) 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 $A \rightarrow \alpha$ 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. $A \rightarrow \varepsilon$
4. $B \rightarrow c$

Step 1. Find all nullable rules and nullable nonterminals:

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

$A \rightarrow B\ C\ D\ \ldots$

where `B,C,D,...` are all nullable non-terminals are nullable rules, and they define nullable nonterminals. 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 non-terminals). 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 .

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(ε) = {}
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 additon, 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↩ 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` ↩ if `A EO S` where `A` represents any nonterminal and `S` represents the starting nonterminal.

For G15 –

`S FB` ↩ 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`)` $\cup$ `Fol(A)`

For G15 –

`Sel(1) = First(ABc) = {b,c}`
`Sel(2) = First(bA) = {b}`
`Sel(3) = First(`ε`) U Fol(A) = {}` $\cup$ `{c} = {c}`

```
Sel(4) = First(c)   = {c}
```

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, below, 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 (see p. 119). For example, this graph shows that the results of steps 3,6, and 8 are needed for step 9.

```
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.

4.3.1 Pushdown Machines for LL(1) Grammars

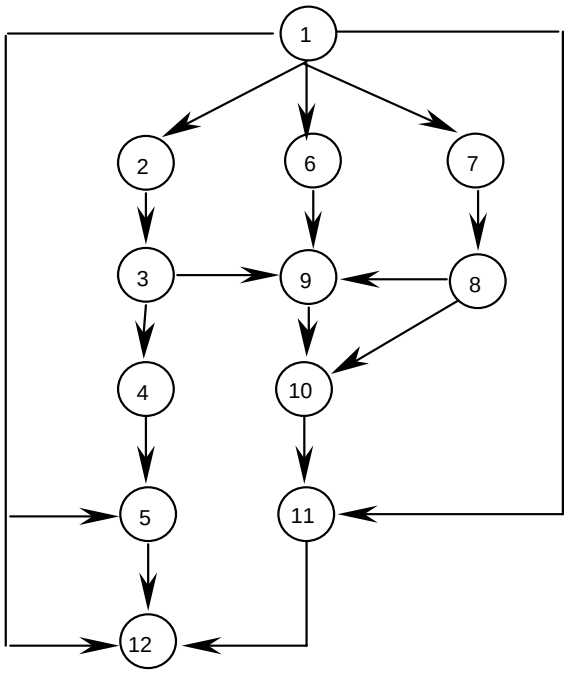


Figure 4.10 Dependency Graph for the Steps in the Algorithm for Finding Selection Sets.

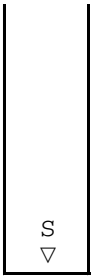
Once the selection sets have been found, the construction of the pushdown machine is exactly as for quasi-simple grammars. For a rule in the grammar, $A \rightarrow \alpha$, fill in the cells in the row for nonterminal `A` and in the columns for the selection set of that rule with `Rep(`α^r`), Retain,` where α^r represents the right side of the rule reversed. For ε rules, fill in `Pop, Retain` in the columns for the 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 $\hookleftarrow$ should contain `Accept`. All other cells are `Reject`. The pushdown machine for grammar G15 is shown in Figure 4.11 (see p. 120).

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 (beginning on page 120):

```
void parse ()
   {   cin >> inp;
```

	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

```
      S ();
      if (inp=='↩') accept; else reject();
   }

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

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

void B ()
   {  if (inp=='c') cin >> inp;        // apply rule 4
```

```
else reject();
}
```

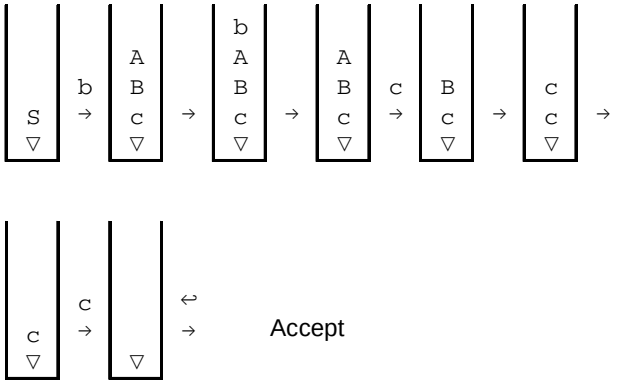
Note that when processing rule 1, an input symbol is not read until a teminal is encountered in the grammar rule (after checking for `a` or `b`, an input symbol should not be read before calling procedure `A`).

Exercises 4.3

Sample Problem 4.3

Show the sequence of stacks that occurs when the pushdown machine of Figure 4.11 (see p. 120) parses the string `bcc↩`.

Solution:



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 → ABD
```

2. A → aA
3. A → ε
4. B → bB
5. B → ε
6. D → dD
7. D → ε

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 don't 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 → Expr + Term
2.  Expr → Term
3.  Term → Term * Factor
4.  Term → Factor
5.  Factor → ( Expr )
6.  Factor → 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 (see p. 118).

```
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    +
```

```
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 → Expr + Term
3.  Term → Term * Factor
```

They are in the form:

$A \rightarrow A\alpha$

Note that any rule in this form cannot be parsed top down. To see this, consider the function 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 “escape hatch”. Any grammar with left recursion cannot be LL(1).

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  →  Aα
A  →  β
```

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  → β R
R  → α R
R  → ε
```

A more detailed and complete explanation of left recursion elimination can be found in Parsons [1992].

This method 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 (see p. 126).

G16:

```
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
```

Note in grammar G16 that an `Expr` is still the sum of one or more `Term`s and a `Term` is still the product of one or more `Factor`s, 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
```

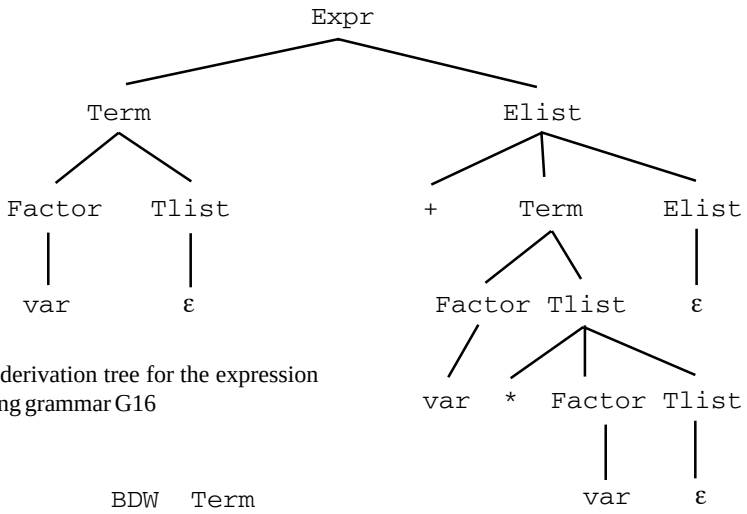


Figure 4.12 A derivation tree for the expression var+var*var using grammar G16

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(ε)                = {}
      4.    First(Factor Tlist)     = {(,var}
      5.    First(* Factor Tlist)   = {*}
      6.    First(ε)                = {}
      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

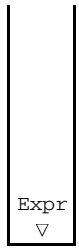
Show a pushdown machine and a recursive descent translator for arithmetic expressions involving addition and multiplication using grammar G16.

Solution:

To build the pushdown machine we make use of the selection sets shown above. 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 on the next page.

We make use of the selection sets again in the recursive descent processor. In each procedure, the input symbols in the selection set tell us which rule of the grammar to apply. Assume that a `var` is represented by the integer 256.

	+	*	(	)	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

```
int inp; const int var = 256;
void Expr ()
   {   if (inp=='(' || inp==var)     // apply rule 1
          {  Term ();
             Elist ();
          }                              // end rule 1
       else reject();
   }
```

```
void Elist ()
   {  if (inp=='+')                    // apply rule 2
         {  cin >> inp;
            Term ();
            Elist ();
         }                             // end rule 2
      else if (inp==')' || inp=='↩') ; // apply rule 3
      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
         {  cin >> inp;
            Factor ();
            Tlist ();
         }                             // end rule 5
      else if (inp=='+' || inp==')' || inp=='↩')
                  ;                    // apply rule 6
      else reject();

void Factor ()
   {  if (inp=='(')                    // apply rule 7
         {  cin >> inp;
            Expr ();
            if (inp==')') cin >> inp;
            else reject();
         }                             // end rule 7
      else if (inp==var) cin >> inp;   // apply rule 8
      else reject();
   }
```

Exercises 4.4

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 functions 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  →   if (Expr) Stmt
2. Stmt  →   while (Expr) Stmt
3. Stmt  →   { StmtList }
4. Stmt  →   Expr ;
```

Write the procedure for the nonterminal `Stmt`. Assume the selection set for rule 4 is `{(, identifier, number}`.

6. Show an *LL(1) grammar* for the language of regular expressions over the alphabet `{0,1}`, and show a *recursive descent parser* corresponding to the grammar.

7. Show how to *eliminate* the *left recursion* from each of the grammars shown below:

(a)
```
1.   A  →  A b c
2.   A  →  a b
```

(b)
```
1.   ParmList  →  ParmList , Parm
2.   ParmList  →  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.

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, or 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:

G19:

```
1.    S → S a B
```

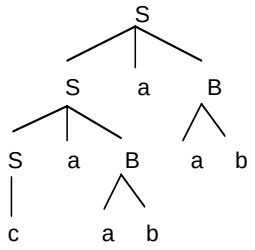


Figure 5.1 A Derivation Tree for the String caabaab, Using Grammar G19

2. S → c
3. B → 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 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:

$$S \Rightarrow \underline{S\ a\ B} \Rightarrow S\ a\ \underline{a\ b} \Rightarrow \underline{S\ a\ B}\ a\ a\ b \Rightarrow$$
$$S\ a\ \underline{a\ b}\ a\ a\ b \Rightarrow \underline{c}\ a\ a\ b\ a\ a\ b$$

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 G20:

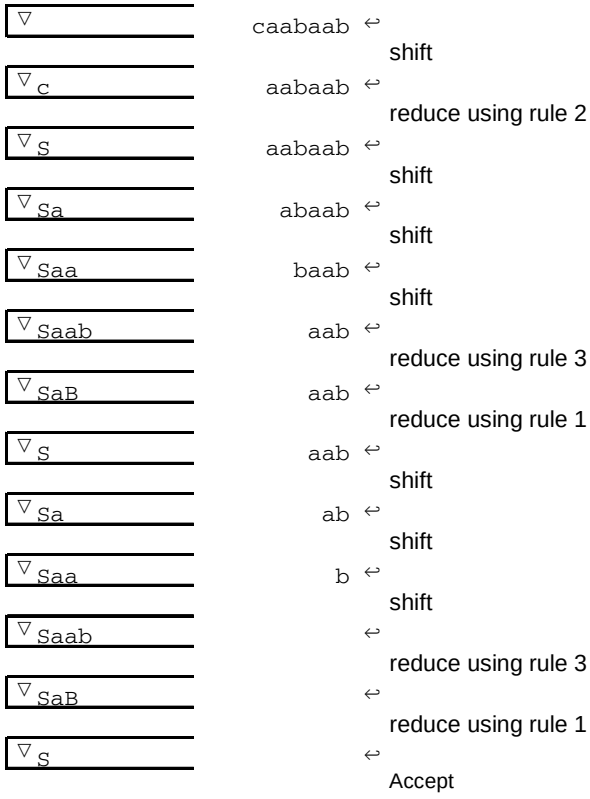


Figure 5.2 Sequence of Stack Frames Parsing `caabaab` Using Grammar G19

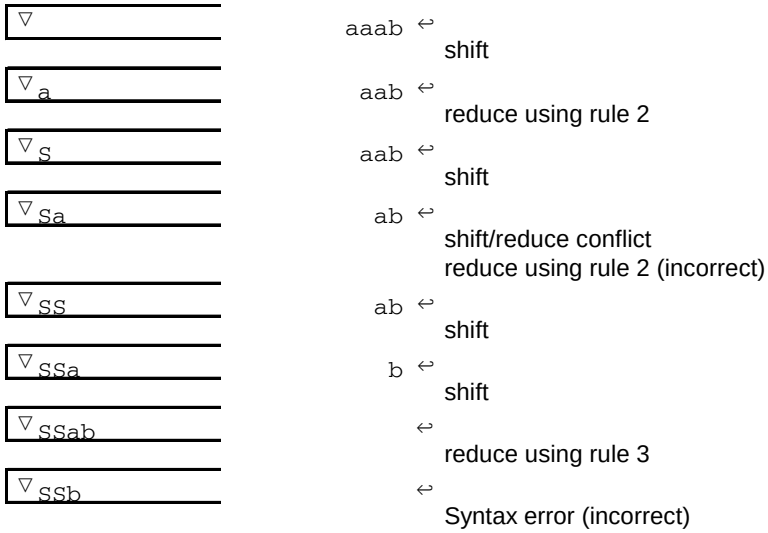


Figure 5.3 An Example of a Shift/Reduce Conflict Leading to an Incorrect Parse Using Grammar G20

G20:

```
1.    S  →  S a B
2.    S  →  a
3.    B  →  a b
```

G21:

```
1.    S  →  S A
2.    S  →  a
3.    A  →  a
```

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 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 G21 is an example of a grammar with a reduce/reduce conflict.

Figure 5.4 shows an attempt to parse the input string `aa` with the shift reduce algorithm, using grammar G21. 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. 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 ambigu-

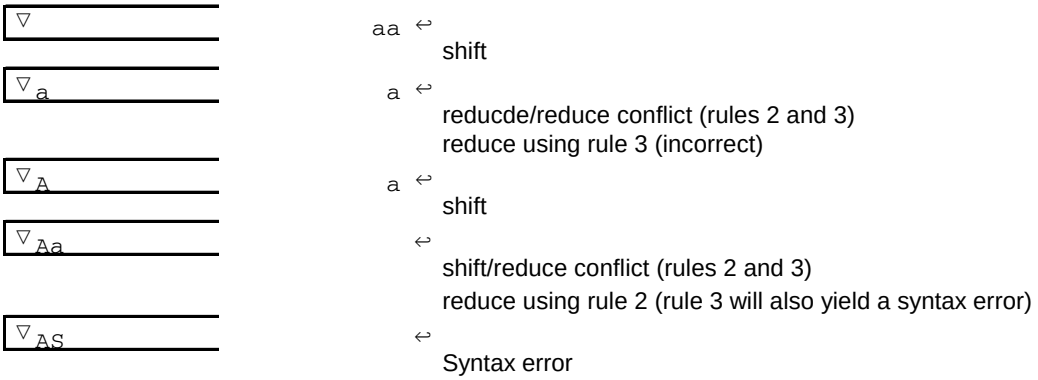


Figure 5.4 A Reduce/Reduce Conflict

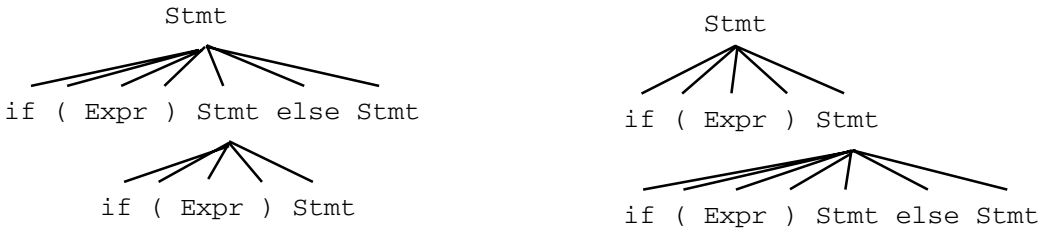


Figure 5.5 Two Derivation Trees for `if (Expr) if (Expr) Stmt else Stmt`

ous 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 (Expr) Stmt else Stmt
2.    Stmt  →  if (Expr) Stmt
```

The Expr in parentheses represents a true (non-zero) or false (zero) condition. Figure 5.5, above, shows two different derivation trees for the statement `if (Expr) if (Expr) Stmt else Stmt`. The second tree shown 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, below, in Figure 5.6.

In this case, the parser will not know whether to treat `if (Expr) 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 it will always find the correct interpretation. Alternatively, the ambiguity may be removed by rewriting the grammar, as shown in Section 3.1.

Stack	Input
▽ ... if (Expr) Stmt	else ... ↩

Figure 5.6 Parser Configuration Before Reading the `else` Part of an `if` Statement

Sample Problem 5.1

Show the sequence of stack and input configurations as the string `caab` is parsed with a shift reduce parser, using grammar G19.

Solution:

▽	caab ↩	shift
▽c	aab ↩	reduce using rule 2
▽S	aab ↩	shift
▽Sa	ab ↩	shift
▽Saa	b ↩	shift
▽Saab	↩	reduce using rule 3
▽SaB	↩	reduce using rule 1
▽S	↩	Accept

Exercises 5.1

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) ▽Sbbc

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 it 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 (Expr) if (Expr) 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 (Expr) Stmt else Stmt
2.   Stmt → if (Expr) 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 method 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
```

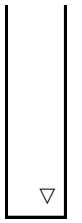
Select the row of the goto table labeled ▽, and the column labeled `B`. If the entry in this cell is `push X`, then the action `reduce 7` would result in the following configuration:

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

	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

being reduced. The stack is initialized with a ▽ symbol, 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.`

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

```
Expr   →  Expr + Term
Expr   →  Term
Term   →  Term * Factor
Term   →  Factor
Factor  →  ( Expr )
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

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 methods 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 method 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 methods are beyond the scope of this text, but are described in Parsons[1992] and Aho [1986].

Exercises 5.2

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 Yacc

For many grammars, the LR parsing tables can be generated automatically from the grammar. One of the most popular software systems that does this is available in the Unix programming environment; it is called ***yacc*** (Yet Another Compiler-Compiler). Recall from Section 1.3.4 that a ***compiler-compiler*** is a program which takes as input the specification of a programming language (in the form of a grammar), and produces as output a compiler for that language. By itself, yacc is really just a parser generator yielding a program which checks for syntax, but since it is possible for the user to augment it with additional features, it can be used to generate a complete compiler. An available public domain version of yacc is called ***bison***. There are also personal computer versions of yacc which use Turbo Pascal as a base language instead of C.

5.3.1 Overview of Yacc

Yacc generates a C function named `yyparse()`, which is stored in a file named `y.tab.c`. This function calls a function named `yylex()` whenever it needs an input token. The `yylex()` function may be written by the user and included as part of the yacc specification, or it may be generated by the lex utility, as described in Section 2.4. A diagram showing the flow of data needed to generate and compile software is shown in Figure 5.9. It assumes that `yylex()` is being generated by lex and that there is a statement in the yacc source file to include the `yylex()` function:

```
#include "lex.yy.c"
```

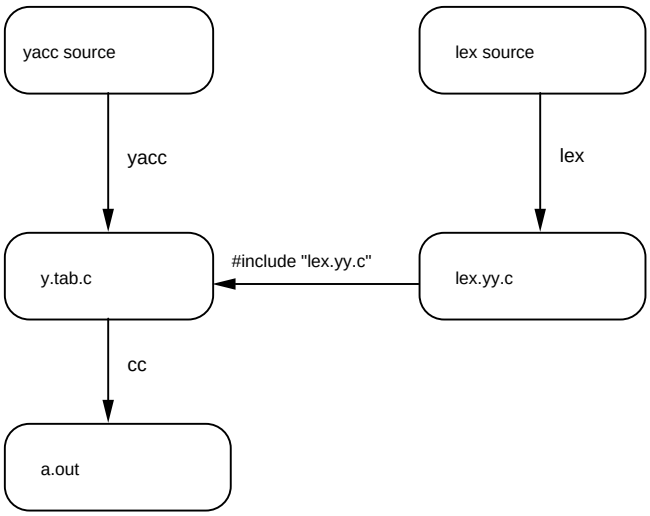


Figure 5.9 Generation and Compilation of Software Using Lex and Yacc

5.3.2 Structure of the Yacc Source File

The input to yacc is called the ***yacc source file***. It consists of three parts, which are separated by the `%%` delimiter:

Declarations
`%%`
Rules
`%%`
Support Routines

The Declarations section contains declarations of token names, stack type, and precedence information which may be needed by yacc. It also may contain preprocessor statements (`#include` or `#define`) and declarations to be included in the output file, `y.tab.c`.

The Rules section is the grammar for the language being specified, such as Pascal. This is the most important part of the yacc source file. Each rule is of the form:

```
nonterminal:    α     {action}
                |β    {action}
                |γ    {action}
                .
                .
                .
                ;
```

where α, β, and γ are definitions of the nonterminal. The vertical bar designates alternative definitions for a non-terminal, as in BNF. An action may be associated with each of the alternatives. This action is simply a C statement which is invoked during the parsing of an input string when the corresponding grammar rule is reduced. The rules may be written in free format, and each rule is terminated with a semicolon.

The third section of the yacc source file contains support routines, i.e., C functions which could be called from the actions in the Rules section. For example, when processing an assignment statement, it may be necessary to check that the type of the expression matches the type of the variable to which it is being assigned. This could be done with a call to a type-checking function in the third section of the yacc source file.

5.3.3 An Example Using Yacc

The purpose of this example is to translate infix expressions involving addition, subtraction, multiplication, division, and unary minus into atoms which represent the primitive operations available on most CPUs. Each atom will consist of up to four parts:

```
(Operation, Left operand, Right Operand, Result)
```

For example, if the input is `(a+b)*c`, the output would be:

```
(ADD, a, b, T1)
(MULT, T1, c,  T2)
```

Whereas if the input is `-a+b*c`, the output would be:

```
(NEG, a, 0, T1)
(MUL, b, c, T2)
(ADD, T1, T2, T3)
```

Note that the unary minus operation produces a `NEG` atom which has only one operand, so the second field is ignored.

In this example, we implement the operands in an atom as character strings. In a real compiler the identifier operands would be implemented as pointers to the symbol table entry for the identifier, and the temporary results (`T1, T2`, etc.) would be implemented as offsets from the top of the run-time stack.

The entire yacc and lex programs for this example are shown below. The essence of the program is in the grammar, which follows the first `%%`. We define the input to be a `List` of `Expr`s, each of which appears on a separate line and may be null. The `Expr` is then defined as we defined it in Section 3.0.3, with grammar G4, only extended to include subtraction, division, and unary operations. This grammar is clearly ambiguous, but yacc permits us to use ambiguous grammars and provides a way of resolving the ambiguity. The actions can refer to the values of the elements of the nonterminal being defined with positional notation. For example, in the following yacc definition:

```
S:    a  S  b  C
```

the value of `a` is referred to as `$1, S` is `$2, b` is `$3`, and `C` is `$4`. The nonterminal being defined may be assigned a value; if so, it is referred to as `$$`. The value of a terminal, or token, is taken from the global variable `yylval`.

Some aspects of this yacc program need to be explained. Beginning with the first Section, the %{ and %} delimiters are used to include C code in the output. They are usually used for macros (`#define`), `#include` statements, and C declarations. The macro `#define YYSTYPE string` is used to declare the type of values to be pushed onto the parse stack. In this case we wish to push strings (actually pointers to char) onto the parse stack, since we are implementing the operations and operands as simple strings. We could not have used `#define YYSTYPE char *`, however, because `YYSTYPE` is used in the resulting C program to declare several variables:

```
YYSTYPE  yylval, yyval;
```

The macro substitution would yield:

```
char * yylval, yyval;
```

which declares `yylval` to be a pointer to char, but `yyval` to be a char, which is incorrect. The use of `typedef` to define a `string` type circumvents this problem. The remainder of the inline C code, as far as the `%}`, is used by the `alloc()` function, described below. We then have some yacc declarations, the first of which, `%token`, is used to declare tokens returned by the `yylex()` function. Tokens are implemented in yacc as integers. Each character corresponds to a one byte integer token. Named tokens, such as `OPERAND` in our example, are then equated to values beginning with 256. The yacc directives `%left` and `%right` are used to declare the precedence of operations (this is how the ambiguity in the grammar is resolved). The directive `%left '+', '-'` means that these operations associate to the left (they are executed from left to right). The fact that the precedence of addition and subtraction is given before the precedence of multiplication and division, indicates that multiplication and division take precedence over addition and subtraction.

The second section of the yacc program includes the grammar for infix expressions. Each time a subexpression generates an intermediate result, we need to allocate a new temporary variable (`T1, T2, T3, ...`). For this purpose we call the `alloc()` function, which returns the next available temporary location in the form of a string (a commercial compiler would return pointers to the locations, or offsets to a stack). Once we know where the result of the operation will be stored we can call the `atom()` function to put out an atom with the appropriate operation and operands. Each time an entire expression has been translated, we need to call the `clear()` function to return allocated storage to the system and to reset counters for the next expression.

The third section of the yacc program includes the main program, with a call to `yyparse()` to start up the translation. This section also includes functions called from our action statements in the second section. In the third section, we also include the output of lex, which is stored in the file `lex.yy.c`. This contains the `yylex()` function which scans the input and returns a lexical token each time it is called. The `yyerror()` function is called when a syntax error is encountered, and it prints the line number which caused the error. Though yacc has the capability of recovering from syntax errors (to scan for more errors), this example does not do so and merely terminates. The `alloc()` function can be called from a lex action as well as from a yacc action. It returns a pointer to an array of three characters which stores the operand in string form, whether it be an identifier (`a,b,c,...`) or a temporary result (`T1,T2,T3...`). These strings are stored in a linked list. The `clear()` function is called when an entire expression has been translated. It frees all nodes in the linked list and resets the counter for temporary locations back to 0. The `atom()` function puts out an atom to `stdout`, using the operation and operands as arguments.

The lex program is shown below:

```
%%
[a-zA-Z]     {yylval = alloc(TOKEN); return OPERAND;}
[ \t]        ;
```

```
.           return yytext[0];
\n          {lineno++; return yytext[0];}
%%
```

The yacc program is shown below:

```
%{
#include <stdio.h>
#include <alloc.h>
typedef char * string;
#define YYSTYPE string        /* type for parse stack */
#define TOKEN 1
#define TEMP 0
int temp = 0;
struct node {
            char name[3];           /* "Tn" or a,b,c,...*/
            struct node * next;     /* linked list */
          };
struct node * tlist = NULL;         /* list head */
char * alloc (int type);
%}
%token OPERAND
%left '+' '-'
%left '*' '/'
%left UMINUS UPLUS
%%
List:                               /* empty list */
            | List '\n'
            | List Expr '\n'  {clear ();}
            ;
Expr:       '+' Expr  %prec UPLUS   {$$ = $2;}
            | '-' Expr  %prec UMINUS  {$$ = alloc(TEMP);
                                       atom ("NEG", $2, 0, $$); }
            | Expr '+' Expr   {$$ = alloc(TEMP);
                               atom ("ADD", $1, $3, $$); }
            | Expr '-' Expr   {$$ = alloc(TEMP);
                               atom ("SUB", $1, $3, $$); }
            | Expr '*' Expr   {$$ = alloc(TEMP);
                               atom ("MUL", $1, $3, $$); }
            | Expr '/' Expr   {$$ = alloc(TEMP);
                               atom ("DIV", $1, $3, $$); }
            | '(' Expr ')'          {$$ = $2;}
            | OPERAND         {$$ = $1;}
            ;
%%
```

```
char *progname;
int lineno = 1;
#include "lex.yy.c"

main (int argc, char *argv[])
{
  progname = argv[0];
  yyparse();
}

yyerror ( char *s)
{
  fprintf(stderr, "%s[%d]: %s\n", progname, lineno, s);
}

char * alloc (int type)
{ static struct node * last = NULL;
  struct node * ptr;
  static char t[3] = "  ";

  ptr = (struct node *) malloc (sizeof (struct node));
  ptr -> next = NULL;
  if (last!=NULL) {last->next = ptr; last = ptr;}
    else tlist = last = ptr;
  if (type==TEMP)
    {
      t[0] = 'T';
      t[1] = '0' + temp++;
      strcpy (last->name, t);
    }
  else strcpy (last->name, yytext);
  return (char *) last;
}

clear ()
/* free up allocated memory, reset counters for next Expr */
{ struct node * ptr;
  while (tlist)
    { ptr = tlist->next;
      free (tlist);
      tlist = ptr;
    }
  temp = 0;
}
```

```
atom (char * operation, char * operand1,
      char * operand2, char * result)
/* put out an atom. */
{
  if (operand2)   /* NEG has only one operand and result */
    printf ("\t%s %s %s %s\n", operation, operand1,
             operand2, result);
  else printf ("\t%s %s %s\n", operation, operand1,
             result);
}
```

In sample Problem 5.3 we attempt to use yacc to build simple expression trees. In this case there are two types which can be pushed on the stack — operators and pointers to nodes in the expression tree. Therefore we need to tell yacc, with a `%union` declaration, that either type may be pushed on the stack:

```
%union {
      struct node * ptr;
      char op;
      }
```

It is then possible to declare a nonterminal to be one of these types, as in the declaration `%type <ptr> Expr` which shows that each `Expr` is a pointer to the subtree which defines that `Expr`. When referring to `yylval`, either in lex or yacc, we now have to specify its type, as `yylval.op`, indicating that it is a single character.

Sample Problem 5.3

Write a yacc program to build simple expression trees. Assume that the operands may be only single letters or single digits and that the operations are addition, multiplication, subraction, and division, with the usual precedence rules. The `yylex()` function may be generated by `lex` or written directly in C. The input should be a single infix expression on each line. After the expression tree is built, dump it to stdout in prefix order. Example:

```
(4+a) * b
      * + 4 a b
4 + a * b
      + 4 * a b
```

Solution:

The lex program is:

```
%%
[a-zA-Z0-9] {yylval.op = yytext[0]; return OPERAND;}
[ \t]             ;
```

```
.                   return yytext[0];
\n                  {lineno++; return yytext[0];}
%%
```

The yacc program is:

```
/* this builds simple expr trees. */
%{
#include <stdio.h>
struct node {
      struct node * left;
      char op;
      struct node * right;
      };
%}
%union {
      struct node * ptr;
      char op;
      }
%token OPERAND
%type <ptr> Expr
%left '+' '-'
%left '*' '/'
%%
List:                           /* empty list */
            | List '\n'
            | List Expr '\n'    {printf ("\t");
                                 prefix ($2);   /* dump expr
                                                   tree */
                                 printf ("\n");}
            ;
Expr:       Expr '+' Expr       {$$ = alloc ($1,'+', $3);}
            | Expr '-' Expr     {$$ = alloc($1, '-', $3);}
            | Expr '*' Expr     {$$ = alloc($1, '*', $3);}
            | Expr '/' Expr     {$$ = alloc($1, '/', $3);}
            | '(' Expr ')'      {$$ = $2;}
            | OPERAND           {$$ = alloc(NULL, yylval.op,
                                      NULL);}
            ;
%%
char *progname;
int lineno = 0;
#include "lex.yy.c"

main (int argc, char *argv[])
```

```
{
  progname = argv[0];
  yyparse();                    /* call the parser */
}

yyerror (char *s)
{
  fprintf(stderr, "%s[%d]: %s\n", progname, lineno, s);
}

struct node * alloc (struct node * left,  char op,
      struct node * right)
/* allocate a node in the expression tree.  Return a pointer
      to the new node */
{ struct node * ptr;

  ptr = (struct node *) malloc (sizeof (struct node));
  ptr->left = left;
  ptr->right = right;
  ptr->op = op;

  return ptr;
}

prefix (struct node * root)
/* print out the exptree in prefix order */
{
  printf ("%c ", root->op);
  if (root->left!=NULL)
      { prefix (root->left);        /* dump left subtree */
        free (root->left); }        /* release memory */
  if (root->right!=NULL)
      { prefix (root->right);       /* dump right subtree */
        free (root->right); }       /* release memory */
  free (root);
}
```

5.3.4 Other Yacc Constructs

We have demonstrated most of the important features of yacc, and the student may find them sufficient to implement parsers for fairly interesting languages, such as the language of arithmetic expressions. However, there are some more advanced constructs which will be needed to generate the parser for MiniC.

The first such advanced feature of yacc has to do with the way yacc handles ***embedded actions***. An embedded action is one which is not at the end of a grammar rule:

```
Nt:   a b {action 1} c d {action 2} ;
```

As we discussed in Section 5.3.3, `action 1` is the embedded action and is counted as one of the items in the rule ($3 in the above example), but we will now need to understand how yacc handles embedded actions. The above rule is actually transformed by yacc into two rules, as Yacc makes up a "fake" nonterminal (which is defined as the null string) for the embedded action. The example, above, would be converted to something like the following:

```
Nt:   a b Fake c d {action 2} ;
Fake: /* null definition */  {action 1} ;
```

Normally, the user need not be aware of this transformation, and everything works fine. However, consider the following example, in which the non-embedded action makes use of a value computed in the embedded action:

```
Nt:   a b { $$ = value;} c d {printf ("%d", $$);} ;
```

In this example we wish to print the value assigned to `$$` in the embedded action. Yacc would convert this rule to two rules:

```
Nt:   a b Fake c d {printf ("%d", $$);} ;
Fake:       {$$ = value;} ;
```

This will not work, since the `$$` referred to in the definition of `Nt` is not the same as the `$$` referred to in the definition of `Fake`. However, we can fix the problem by referring to the value of the embedded action itself:

```
Nt:   a b { $$ = value;} c d {$$ = $3; printf ("%d", $$);} ;
```

which works because yacc transforms it into the following two rules:

```
Nt:   a b Fake c d {$$ = $3;  printf ("%d", $$);} ;
Fake:       {$$ = value;} ;
```

The fake nonterminals are assigned numbers by yacc, so if you should see error messages referring to nonterminals, such as `$$23`, for example, it is probably a nonterminal that yacc made up to represent an embedded action.

If the parser stack has been declared to be a union of several types, the type of the embedded action can be specified by placing the type between the `$` and the item number. For example, if the type of the embedded action in the above action were labels, we would refer to it as `$<labels>3`.

The other advanced feature of yacc, which may be needed in constructing compilers, has to do with ***inherited attributes***. Up to this point, all values that have been

passed from one rule to another (via an assignment to `$$`) have been passed in a direction corresponding to an upward direction in the derivation tree. As we saw in Section 4.6, these values are called ***synthesized attributes***. We now wish to consider the problem of passing values between rules in a direction which corresponds to a downward direction in the derivation tree; i.e., we wish to use inherited attributes in our yacc grammar.

Suppose, for example, that the source language includes a simplified `Switch` statement, defined as follows:

```
SwitchStmt: switch ( Expr ) { CaseList }
          ;
CaseList:   case NUM : StmtList
          | CaseList case NUM : StmtList
```

We will need to put out `TST` atoms which compare the `Expr` in the `SwitchStmt` with each of the `NUM`s in the `CaseList`, so an attribute representing the runtime location of the `Expr` must be passed (down the tree) from `SwitchStmt` to the `CaseList`s, hence the need for inherited attributes. When writing actions in the definition of `CaseList`, we can assume that the preceding items in the `SwitchStmt` are on the parser stack, and they can be referred to with negative item numbers as follows:

```
      SWITCH (   Expr )   { CaseList  }
      $-4    $-3 $-2  $-1 $0
```

Thus the action for `CaseList` might be:

```
CaseList: case NUM ':' {TST atom which compares $-2 with
            $2} StmtList
      | CaseList case NUM ':' {TST atom which compares $-2
        with $3}  StmtList
```

– in which `$-2` represents the location of the result of the `Expr` which is to be compared with a `NUM` in the `CaseList`. These and other advanced features of yacc are explained in fine detail in Levine [1992].

Exercises 5.3

1. Which of the following input strings would cause this yacc and lex program to produce a *syntax error* message?

```
/* yacc program */
%%
line:     s '\n'
     ;
```

```
s:   'a' s 'b'
     | 'b' a 'c'
     ;
a:   'b' a 'b'
     | 'a' 'c'
     ;
%%
#include "lex.yy.c"
main ()
{ yyparse();}

/* lex program */
%%
.    return yytext[0];
\n   return yytext[0];
```

(a) `bacc` (b) `ab` (c) `abbacbcb`
(d) `bbacbc` (e) `bbacbb`

2. Show the *output* produced by each of the input strings given in Problem 1, using the yacc and lex programs shown below.

```
/* yacc program */
%%
line:     s '\n'
     ;
s:   'a' s 'b' {printf ("rule 1\n");}
     | 'b' a 'c'    {printf ("rule 2\n");}
     ;
a:   'b' a 'b' {printf ("rule 3\n");}
     | 'a' 'c' {printf ("rule 4\n");}
     ;
%%
#include "lex.yy.c"
main ()
{ yyparse();}
```

```
/* lex program */
%%
.    return yytext[0];
\n   return yytext[0];
%%
```

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,` 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 *yacc grammar* (not a complete yacc program, but just the part after the first `%%`) that defines a *Sexpr*. Assume that the `yylex()` function returns either `ATOM, (, )`, or `.`, where `ATOM` represents any atom.

(b) Show a *yacc grammar* (again, not a complete yacc program) that defines a *List*. Assume that the `yylex()` function returns either `ATOM, NIL, (, ),` or `.`, where `ATOM` represents any atom other than `NIL.`

(c) Add *actions* 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 yacc and lex 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 lex and yacc programs 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
     2.032000
2+3*5/2
     9.500000
(2+3)*5/2
     12.500000
16/(2*3 - 6*1.0)
     division by zero
```

Unfortunately, the program as shown below does not work. It contains four mistakes, some of which are syntactic lex or yacc errors; some of which are syntactic C language 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.

```
/* lex program */
%%
[0-9]+\.?[0-9]*[eE][+-]?[0-9]+?   {sscanf
                    (yytext,"%f",&yylval);
                    return NUM; }
[    ]                   ;
```

```
"\n"                        return  yytext[0];
.                           return yytext[0];
%%

/* yacc program */
%token NUM
%{
#define YYSTYPE float
%}
%left "+"  "-"
%left "*" "/"
list:
     | list expr '\n'     {printf ("%f\n",
                          $2);}
     ;
expr:       expr '+' expr     {$$ = $1 +
                              $3;}
     | expr '-' expr          {$$ = $1 -
                              $3;}
     | expr '*' expr          {$$ = $1 *
                              $3;}
     | expr '/' expr          {if ($3=0)
          yyerror ("division by zero\n");
          else $$ = $1 / $3;}
     | '(' expr ')'           {$$ = $1;}
     | NUM                    {$$ = $1;}
     ;
%%
#include "lex.yy.c"

yyerror (char *s)
{ printf ("%s: %s\n", s, yytext); }
```

6. Show lex and yacc programs which will check for proper syntax of regular expressions over the alphabet {0,1}. Examples:

`(0 + 1)* . (0 + 1*)`	Syntax is ok
`((0 + 1)*`	Syntax is not ok

5.4 Arrays

Although arrays are not included in our definition of MiniC, 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 [1992] or Aho [1986].

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 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[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 C++. For example, if `V` were declared to contain twenty elements, `char V[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`. A vector maps directly to the computer's memory.

Now we introduce a second dimension; suppose `M` is declared as a matrix, or two-dimensional array, `char M[10][15];`. A reference to an element of this array will compute an offset of fifteen elements for each row after the first. Also, we must add to

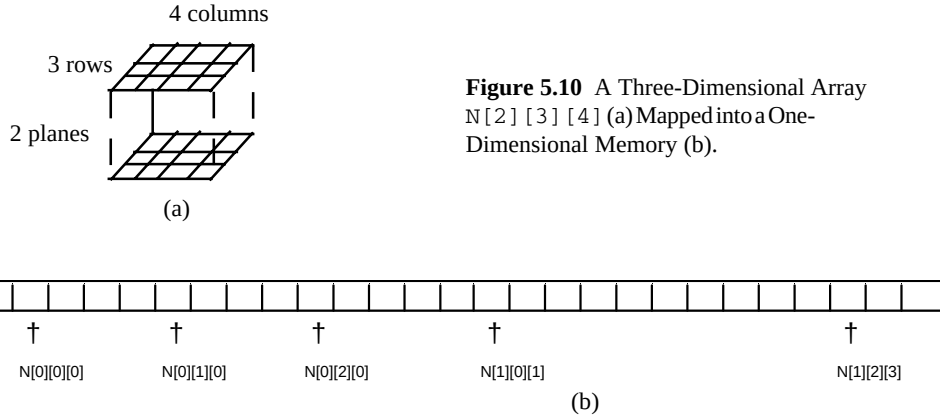


Figure 5.10 A Three-Dimensional Array `N[2][3][4]` (a) Mapped into a One-Dimensional Memory (b).

this offset the number of columns 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[Rows][Cols]`, the formula for the offset of `M[R][C]` is `R*Cols + C`.

For a three-dimensional array, `char A[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, `A[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 $A[D_1][D_2][D_3]\ldots[D_n]$, then a reference to $A[S_1][S_2][S_3]\ldots[S_n]$ will require an offset computed by the following formula:

$$S_1*D_2*D_3*D_4*\ldots*D_n + S_2*D_3*D_4*\ldots*D_n + S_3*D_4*\ldots*D_n \quad +\ldots$$
$$+ S_{n-1}*D_n + S_n.$$

In this formula, D_i represents the number of elements in the `i`th dimension and S_i represents the `i`th subscript in a reference to the array. Note that in some languages, such as 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, assuming that arrays must be dimensioned with constants, the product of dimensions `D2*D3*D4` 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 S_i 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 added rules 8-10.

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 G23, 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 `Factor` in that rule.

Grammar G23:

7. $\texttt{Factor}_{e} \rightarrow \texttt{var}_{v}\ \{\texttt{MOV}\}_{0,,sum}\ \texttt{Subs}_{v,sum,i}$
   `e ← v[sum]`
   `i ← 1`
   `sum ← Alloc`

8. $\texttt{Subs}_{v,sum,i1} \rightarrow \texttt{[}\ \texttt{Expr}_{e}\ \texttt{]}\ \{\texttt{MUL}\}_{e,=D,T}\ \{\texttt{ADD}\}_{sum,T,sum}\ \texttt{Subs}_{v,sum,i2}$
   `D ← Prod(v,i1)`
   `i2 ← i1 + 1`
   `T ← Alloc`

9. $\texttt{Subs}_{v,sum,i} \rightarrow \{\texttt{check}\}_{i,v}$

The nonterminal `Subs` has three attributes: `v` (inherited) represents a pointer 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 function `Prod(v,i)`. This function 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 ← 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 $\{\texttt{check}\}_{i,v}$ in rule 9. This action symbol represents a

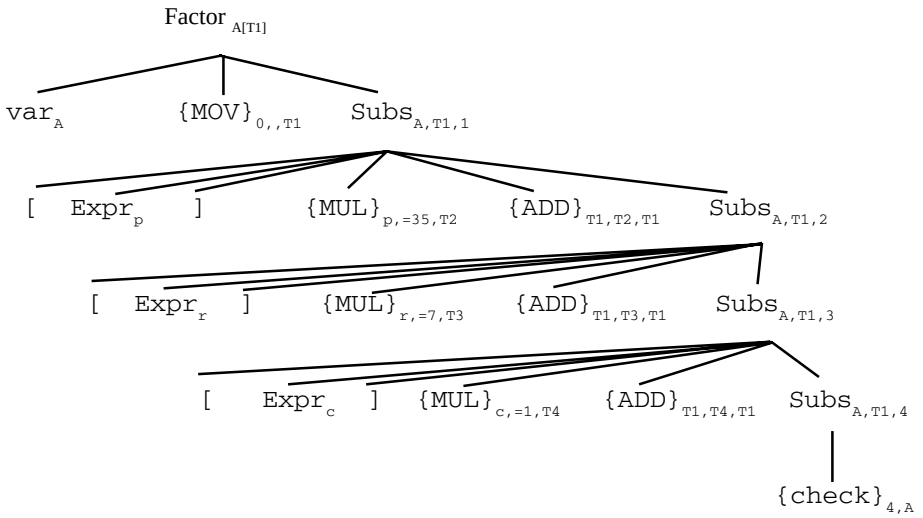


Figure 5.11 A Derivation Tree for the Array Reference `A[p][r,][c]`, Which is Declared as `int A[3][5][7].`

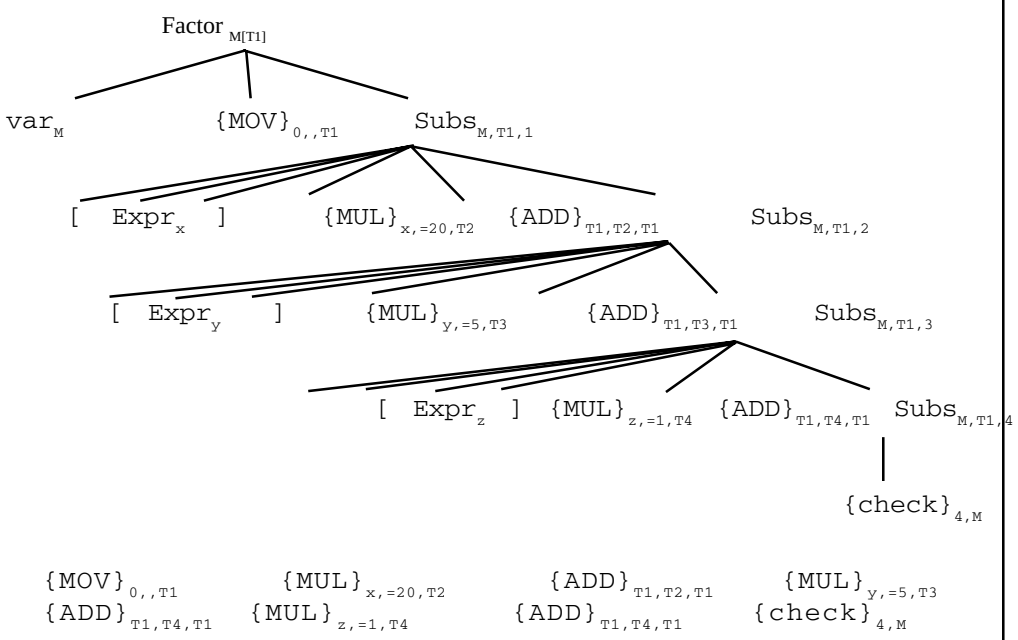
procedure call, not an atom. The purpose of the procedure is to compare the number of dimensions of the variable, `v`, as stored in the symbol table, with the value of `i`, the number of subscripts plus one. The `check(i,v)` procedure simply puts out an error message if the umber of subscripts does not 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[3][5][7]`. An attributed derivation tree for the reference `A[p][r][c]` is shown, above, 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 atttributes 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

Assume the array `M` has been declared to have two planes, four rows, and five columns (`int M[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 5.11. Also show the sequence of atoms which would be put out as a result of this array reference.

Solution:



Exercises 5.4

1. Assume the following array declarations:

```
int V[13];
int M[12][17];
int A3[15][7][5];
int Z[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 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. 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 *C++ function* –

```
int offSet (int D[max], int S[max]);
```

that computes the offset for an array reference $A[S_0][S_1] \ldots [S_{max-1}]$ where the array has been declared as $\texttt{char } A[D_0][D_1] \ldots [D_{max-1}]$.

(b) Improve your C++ function, if possible, to minimize the number of run-time multiplications.

5.5 Case Study: Syntax Analysis for MiniC

In this section we continue the development of a compiler for MiniC, a small subset of the C++ programming language. We do this by implementing the syntax analysis phase of the compiler using the yacc utility as described in Section 5.3, above. The parser generated by yacc will obtain input tokens by calling `yylex()` (the function generated by lex as described in Section 2.5). The parser will then check the tokens for correct syntax.

In addition, we provide supporting functions which enable 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 MiniC compiler, syntax analysis and semantic analysis have been combined into one program.

The complete yacc source file is shown in Appendix B.2 and is explained in the following sections. The input to yacc is the file `MiniC.y` (divided into three sections by `%%` delimiters), and the output, stored in the file `y.tab.c`, is the `yyparse()` function. The `MiniC.y` file contains the main program (in the third section) which receives control when the compiler is invoked. It calls the `yyparse()` function, which in turn calls the `yylex()` function when it needs a token. As output, it produces a file of atoms. It then invokes the code generator `code_gen()` if no errors have been detected. Consequently, the program generated by yacc will ultimately contain all three phases of the compiler (the lexical phase and the code generator are brought in with `include` directives).

5.5.1 Header Files Included

Our yacc source file begins with three `include` statements enclosed in %{ and %} braces, indicating that these directives are to be copied verbatim to the output file:

```
#include <stdio.h>
#include "mini.h"
#include "MiniC.h"
```

The first `include` is to support the necessary input/output function calls. The `mini.h` header file contains declarations which are needed for the back end of the compiler (code generation), but not the front end (lexical and syntax analysis).

The third header file included, `MiniC.h`, contains declarations which are needed by the front end of the compiler. It specifies constants determining the maximum size of the symbol table and the table of address labels. It also defines a type `ADDRESS`, which is intended to be a run-time address on the target machine. This header file also defines structures for the lexical tables, as described in Section 2.5.

The parser produces a file of atoms, the structure of which is defined in the `MiniC.h` header file. This record structure is shown in Figure 5.12. The remainder of the `MiniC.h` header file defines the atom classes which are not also used as machine op

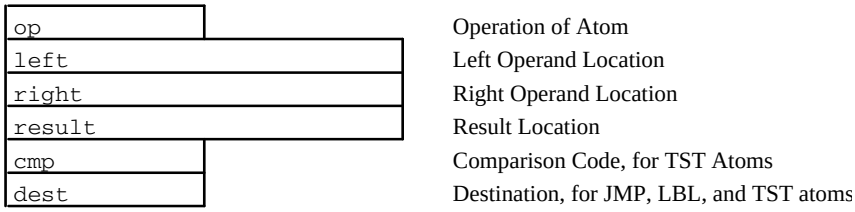


Figure 5.12 Record Structure of the File of Atoms

codes (these are defined in the `mini.h` header file because they are needed to compile the Mini machine simulator).

5.5.2 Other Declarations in the First Section

The parser stack needs to store three kinds of data: target machine addresses (`address`), integer codes for comparisons and type declarations (`code`), and label numbers (`labels`). We tell this to yacc with a `%union` statement:

```
%union      {
            ADDRESS address;
            int code;       /* comparison code 1-6 */
            struct          {int L1;
                             int L2;
                             int L3; } labels;
            }
```

This is similar in meaning to the `union` declaration in C; in effect, it allows the user to bind different interpretations (types) to the same storage location. When assigning a value to a syntactic type in the rules section, we can refer to any of these fields to indicate the type of the value pushed on the parser stack. For example, in the following rule:

```
Type:   terminal            {$$.code = $1.L1;}
        ;
```

the value assigned to the `Type` being defined (`$$`) is the type of `code`, whereas the type of the terminal (`$1`) is the type of `L1`. Notice that we did not need to give the structure name, `labels`, in the reference to `$1.L1`. The reason is that yacc fills in the structure name for us.

The lexical tokens are declared with `%token` statements. We must declare a type for those tokens which have a value part (as one of the types listed in the `%union` statement) so the parser will know which type is being pushed on the stack. For ex-

ample, since an identifier's value part is it's target machine address, the declaration is `%token <address> IDENTIFIER`.

Action statements in the grammar can also push values onto the parser stack for nonterminals. In this case, we must again inform yacc of the type to be associated with these nonterminals, using the `%type` statement. For example, the nonterminal `Type` receives the value 1 if the type is `INTEGER`, and 2 if the type is `REAL`. We use the `code` stack type for this purpose, and consequently declare the `Type` nonterminal: `%type <code> Type`.

The first section then concludes with precedence information for the parsing of arithmetic expressions. The %right '=' directive indicates that the assignment operator associates to the right (e.g. `a = b = 0` should assign `0` to `b`, and then `0` to `a`). The `%left COMPARISON` directive, `%left '+' '-'` directive, and the `%left '*' '/'` directives all indicate that these operations associate to the left. The fact they are listed in separate statements specify that '*' and '/' take precedence over '+' and '-', which take precedence over `COMPARISON` operators, which take precedence over the assignment operator. Finally, the `%left UMINUS UPLUS` directive indicates that those unary operations are also ***left associative*** and take precedence over multiplication and division.

5.5.3 The Rules Section

The second section of the yacc source file contains the grammar for MiniC, together with action statements that generate the appropriate atoms. Each call to the `atom()` function writes an atom to the file of atoms. The most difficult part of this section has to do with the processing of labels for the various control structures. To see why it is necessary to use the parser stack for these labels, consider a nested `while` statement:

```
while (x>y) while (a!=0) Stmt
```

This shows that it is not possible to generate a label (`L2`) after the key word `while` is seen, and refer to it as `L2` after the `Stmt` is seen. Because these control structures can be nested, it is necessary to use the parser stack so that `JMP` atoms can be associated with the proper destination `LBL` atoms. The definition of a `while` statement, with actions, is shown below:

```
WhileStmt:    WHILE        {$$.L1 = newlabel();
                            atom (LBL,NULL,NULL,NULL,0,$$.L1);}
              '(' Expr ')' {$$.L2 = newlabel();
                            atom (TST,$4, zero, NULL,1,$$.L2);}
                  Stmt     {atom (JMP,NULL,NULL,NULL,0,
                            $<labels>2.L1);
                            atom (LBL,NULL,NULL,NULL,0,
                            $<labels>6.L2);}
              ;
```

In order to understand these actions the reader should refer back to Figure 4.17(a) in which the control structure of a `while` statement is diagrammed. The first atom put out is a `LBL` atom in which the destination field is a new label (`L2`). We can then refer to this in another action, but must be careful to refer to the correct grammar item (recall that actions are included when numbering the items in a grammar statement: `$1, $2, $3`, ...). The assignment to `$$.L2` is done in the second item, so we refer to it below as `$2.L2`. Since `$2` is an embedded action, we need to specify its type as `$<labels>2.L1`. After the `')'` is seen, a `TST` atom is put out, which compares the value of the `Expr` with 0 (false). If the `Expr` is 0, it jumps out of the loop (to label `L2`). After the `Stmt`, which is the body of the loop, we put out a `JMP` atom to label `L1`, which is at the beginning of the loop.

The other control structures are handled in a similar fashion, and should be compared with Figure 4.17. If the student attempts to generate the parser for MiniC, there will be an error message from yacc: shift/reduce conflict. This error results from the ambiguous `if else` statement in the MiniC grammar. Fortunately, yacc resolves the ambiguity with a shift whenever possible, which means that all `else` clauses are matched with the closest previous unmatched `if`. In other words, the user need not be concerned about this shift/reduce error message.

Arithmetic expressions are handled with a grammar similar to grammar G4, which is ambiguous. Fortunately the ambiguity is resolved with the `%left` precedence directives described in Section 5.5.2, above.

In the processing of arithmetic expressions, each operation invokes `alloc(1)` to reserve temporary storage for the result of the operation. These temporary storage locations are never released or reused for other purposes, simply to keep the compiler short and simple. The student is encouraged to write a `release()` function which would return unused temporaries. One approach would be to locate these items after the program instructions, rather than in low memory locations, using high memory as a run-time stack. Each time a complete expression is evaluated, its result could be moved to the lowest stack location no longer needed.

5.5.4 Supporting Functions

The third section of the yacc program contains the main program as well as supporting functions. The main program opens the output file `atoms` and then installs the constants 1.0 and 0.0(which are used as logical values True and False) in the table of constants before invoking the parser `yyparse()`.

The `yyerror()` function is called by `yyparse()` when there is a syntax error, and processing terminates. Yacc does have the capability of recovering from syntax errors, but that is beyond our scope. The `yyerror()` function is also called from `yylex()` when undeclared identifiers or multiply-declared identifiers are encountered. In this case, the program continues to look for additional errors, but no code is generated.

The most important supporting function is the `atom()` function, which generates an atom as a record in the file `atoms`, given the operation, the left and right operands, the result, the comparison code (for `TST` atoms), and the destination label

number (for `TST` and `JMP` atoms). The complete source input to yacc, in the file `miniC.y`, is shown in appendix B.

Exercises 5.5

1. Extend the MiniC language to include a `do` statement defined as:

   ```
   DoStmt → do Stmt while '(' Expr ')' ';'
   ```

   Modify the files `MiniC.l` and `MiniC.y`, 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 `Expr` are already defined. The tokens `do` and `while` need to be defined. 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.

2. Extend the MiniC language to include a `switch` statement defined as:

   ```
   SwitchStmt → switch '(' Expr ')' CaseList
   CaseList → case number ':' Stmt CaseList
   CaseList → case number ':' Stmt
   ```

   Modify the files `MiniC.l` and `MiniC.y`, shown in Appendix B, so that the compiler puts out the correct *atom* sequence implementing 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 MiniC language to include initializations in decalarations, such as:

```
int x=3, y, z=0;
```

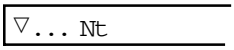
Modify the files `MiniC.l` and `MiniC.y`, shown in Appendix B, so that the compiler puts out the correct *atom* sequence implementing this feature. You will need to put out a `MOV` atom to assign the value of the constant to the variable.

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:

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



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 Unix utility, *yacc*, 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. Lex, the utility for generating lexical scanners (Section 2.4), and yacc are designed to work together.

Finally we looked at an *implementation of MiniC*, our case study language which is a subset of C++, using yacc. This compiler works with the lexical phase discussed in Section 2.4 and is shown in Appendix B.3.

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 [1990]). 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 don't 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, p. 23).

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 $\mathbf{C}^{\texttt{Pas} \rightarrow \texttt{Mac}}_{\texttt{Pas}}$ and $\mathbf{C}^{\texttt{Pas} \rightarrow \texttt{Mac}}_{\texttt{Mac}}$, the compiler we want is $\mathbf{C}^{\texttt{Ada} \rightarrow \texttt{Mac}}_{\texttt{Mac}}$, and each is composed of two parts, as shown in Figure 6.1. We write $\mathbf{C}^{\texttt{Ada} \rightarrow \texttt{Atoms}}_{\texttt{Pas}}$ 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 (a) and link it with the back end of our Pascal compiler to form a complete Ada compiler for the Mac, as shown in Figure 6.2 (b).

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. 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.

We have the source code for a Pascal Compiler:

$$C^{\text{Pas} \rightarrow \text{Mac}}_{\text{Pas}} = C^{\text{Pas} \rightarrow \text{Atoms}}_{\text{Pas}} + C^{\text{Atoms} \rightarrow \text{Mac}}_{\text{Pas}}$$

We have the Pascal compiler which runs on the Mac:

$$C^{\text{Pas} \rightarrow \text{Mac}}_{\text{Mac}} = C^{\text{Pas} \rightarrow \text{Atoms}}_{\text{Mac}} + C^{\text{Atoms} \rightarrow \text{Mac}}_{\text{Mac}}$$

We want an Ada Compiler which runs on the Mac:

$$C^{\text{Ada} \rightarrow \text{Mac}}_{\text{Mac}} = C^{\text{Ada} \rightarrow \text{Atoms}}_{\text{Mac}} + C^{\text{Atoms} \rightarrow \text{Mac}}_{\text{Mac}}$$

We write the front end of the Ada compiler in Pascal:

$$C^{\text{Ada} \rightarrow \text{Atoms}}_{\text{Pas}}$$

Figure 6.1 Using a Pascal Compiler to Construct an Ada Compiler

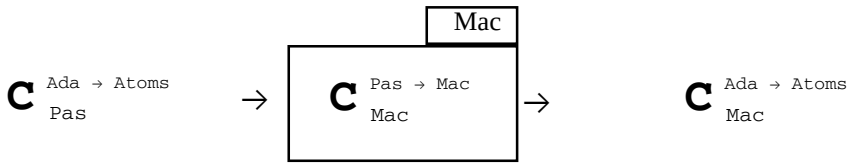


Figure 6.2 (a) Compile the Front End of the Ada Compiler on the Mac

$$C^{\text{Ada} \rightarrow \text{Atoms}}_{\text{Mac}} + C^{\text{Atoms} \rightarrow \text{Mac}}_{\text{Mac}} = C^{\text{Ada} \rightarrow \text{Mac}}_{\text{Mac}}$$

Figure 6.2 (b) Link the Front End of the Ada Compiler with the Back End of the Pascal Compiler to Produce a Complete Ada Compiler.

Sample Problem 6.1

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.

Solution:

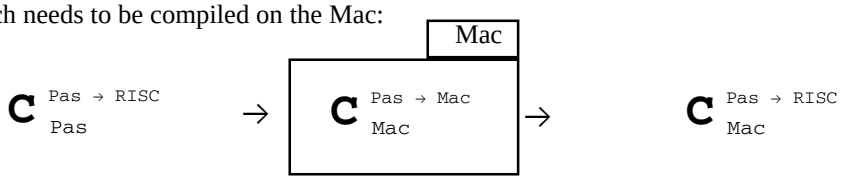
We want $C^{\text{Ada} \rightarrow \text{RISC}}_{\text{RISC}}$

Write (in Pascal) the back end of a compiler for the RISC machine:

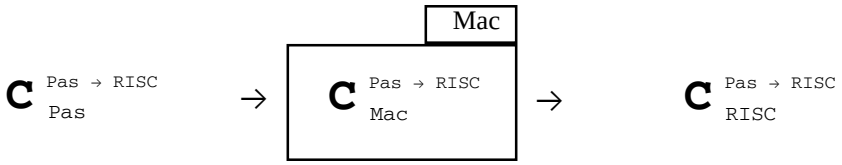
$C^{\text{Atoms} \rightarrow \text{RISC}}_{\text{Pas}}$

We now have $C^{\text{Pas} \rightarrow \text{RISC}}_{\text{Pas}} = C^{\text{Pas} \rightarrow \text{Atoms}}_{\text{Pas}} + C^{\text{Atoms} \rightarrow \text{RISC}}_{\text{Pas}}$

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 memory and compile again:



and the output is the compiler that we wanted to generate.

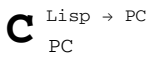
Exercises 6.1

1. Show the big C notation for each of the following compilers (assume that each uses an intermediate form called “Atoms”):

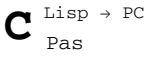
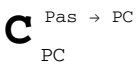
(a) The back end of a compiler for the Vax.

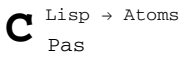
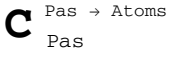
(b) The source code, in Pascal, for a COBOL compiler whose target machine is the PC.

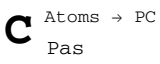
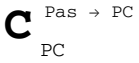
(c) The souce code, in Pascal, for the back end of a FORTRAN compiler for the Vax.

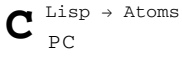
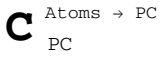
2. Show how to generate 

without writing any more programs, given a PC machine and each of the following collections of compilers:

(a)  

(b)  

(c)  

3. Given a NeXT computer and the following compilers, show how to generate a Pascal (Pas) compiler for the MIPS machine without doing any more programming. (Unfortunately, you can't afford to buy a MIPS computer.)

$$\mathbf{C}^{\texttt{Pas} \rightarrow \texttt{NeXT}}_{\texttt{Pas}} = \mathbf{C}^{\texttt{Pas} \rightarrow \texttt{Atoms}}_{\texttt{Pas}} + \mathbf{C}^{\texttt{Atoms} \rightarrow \texttt{NeXT}}_{\texttt{Pas}}$$

$$\mathbf{C}^{\texttt{Pas} \rightarrow \texttt{NeXT}}_{\texttt{NeXT}} = \mathbf{C}^{\texttt{Pas} \rightarrow \texttt{Atoms}}_{\texttt{NeXT}} + \mathbf{C}^{\texttt{Atoms} \rightarrow \texttt{NeXT}}_{\texttt{NeXT}}$$

$$\mathbf{C}^{\texttt{Atoms} \rightarrow \texttt{MIPS}}_{\texttt{Pas}}$$

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, below, in Figure 6.3; 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 6.3 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

The C++ statement `If (A>B) A = B * C` might result in the following sequence of atoms:

```
(MOV, 1,,T1)              // Move True into T1
(TST, A, B,, 3, L1)       // Branch to L1 if A>B
(MOV 0,, T1)              // Move False into T1
(LBL, L1)
```

```
(TST, T1, 0, 1, L2)           // Branch to L2 if T1==0
(MUL, B, C, A)
(JMP, L3)
(LBL, L2)
(LBL, L3)
```

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,='1'           // Load 1 into Reg. R1
      STO   R1,T1             // Store it in T1
      LOD   R1,A
      CMP   R1,B,3            // Compare A > B ?
      JMP   L1                // Branch if true
      LOD   R1,='0'           // Load 0 into Reg. R1
      STO   R1,T1             // Store it in T1
L1:
      LOD   R1,T1
      CMP   R1,='0',1         // Compare T1==0?
      JMP   L2                // Branch if true
      LOD   R1,B
      MUL   R1,C              // B * C
      STO   R1,A              // A = B * C
      CMP   0,0,0
      JMP   L3                // Unconditional Branch
L2:
L3:
```

Exercises 6.2

1. For each of the following C++ 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)
(MOV, 1,, T1)
(TST, i, 10,, 4, L2)     // Is i<=10 ?
(MOV, 0,, T1)            // No, result is false
(LBL, L2)
(TST, T1, 0,, 6, L3)     // Branch if not false
(JMP, L4)
(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)
(MOV, 1, T2)
(TST, a, T1,, 6, L1)     // Is a!=T1 ?
(MOV, 0, T2)
(LBL, L1)
(TST, T2, 0, , 1, L2)    // Is T2 false ?
```

```
(MOV, 0, a)                 // a = 0
(JMP, L3)
(LBL, L2)
(ADD, b, 3, T2)             // T2 = b + 3
(MOV, T2,, b)               // b = T2
(LBL, L3)
```

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 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, below, in Figure 6.4 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.

```
Atom                    Location    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   ?
(LBL, L1)                                       (L1 = 9)
```

Figure 6.4 Problem in Generating a Jump to a Forward Destination

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 6.5, below, for the same atom sequence shown in Figure 6.4. 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 changed, using the Fixup table and the Label table, to 9.

				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 6.5 Use of the Fixup Table and Label Table in a Single Pass Code Generator for Forward Jumps

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 6.6 which, again, uses the atom sequence given in Figure 6.4.

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

The following atom string resulted from the C++ 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. Assume that 0 is

```
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 6.6 Forward Jumps Handled by a Multiple Pass Code Generator

stored at memory location 0, and 1 is stored at memory location 1. Your object code should begin at location 2.

```
(LBL, L1)
(MOV, 1,, T1)
(TST, i, x,, 4, L2)            // Is i<=x ?
(MOV, 0,, T1)
(LBL, L2)
(TST, T1, 0,, 1, L3)           // Branch if T1 is false
(ADD, x, 2, x)
(MUL, i, 3, i)
(JMP, L1)                      // Repeat the loop
(LBL, L3)                      // End of loop
```

Solution:

(1) Single Pass

			Fixup Table		Label Table	
Atom	Loc	Instruction	Loc	Label	Label	Value
(LBL, L1)					L1	2
(MOV, 1,, T1)	2	LOD R1,1				
	3	STO R1,T1				
(TST,i,x,,4,L2)	4	LOD R1,i				
	5	CMP R1,x,4				
	6	JMP 0	6	L2		
(MOV, 0,, T1)	7	LOD R1,0				
	8	STO R1,T1				
(LBL, L2)					L2	9
(TST,T1,0,,1,L3)	9	LOD R1,T1				
	10	CMP R1,0,1				
	11	JMP 0	11	L3		
(ADD, x, 2, x)	12	LOD R1, x				
	13	ADD R1,='2'				
	14	STO R1,x				
(MUL, i, 3, i)	15	LOD R1,i				
	16	MUL R1,='3'				
	17	STO R1,i				
(JMP, L1)	18	CMP 0,0,0				
	19	JMP 2				
(LBL, L3)					L3	20
...						
	6	JMP 9				
	11	JMP 20				

(2) Multiple passes

			Label Table	
Atom	Loc	Instruction	Label	Value
Begin First Pass:				
(LBL, L1)			L1	2
(MOV, 1,, T1)	2-3			
(TST,i,x,, 4,L2)	4-6			
(MOV, 0,, T1)	7-8			
(LBL, L2)			L2	9
(TST,T1,0,,1,L3)	9-11			

```
(ADD, x, 2, x)     12-14
(MUL, i, 3, i)     15-17
(JMP, L1)          18-19
(LBL, L3)                                        L3     20

Begin Second Pass:
Atom                 Loc     Instruction

(LBL, L1)
(MOV, 1,, T1)        2       LOD  R1,1
                     3       STO  R1,T1
(TST,i,x,,4,L2)      4       LOD  R1,i
                     5       CMP  R1,x,4
                     6       JMP  9
(MOV, 0,, T1)        7       LOD  R1,0
                     8       STO  R1,T1
(LBL, L2)
(TST,T1,0,,1,L3)     9       LOD  R1,T1
                     10      CMP  T1,0
                     11      JMP  20
(ADD, x, 2, x)       12      LOD  R1,x
                     13      ADD  R1,='2'
                     14      STO  R1,x
(MUL, i, 3, i)       15      LOD  R1,i
                     16      MUL  R1,='3'
                     17      STO  R1,i
(JMP, L1)            18      CMP  0,0,0
                     19      JMP  2
(LBL, L3)
```

Exercises 6.3

1. The following atom string resulted from the C++ 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
```

```
(MOV, 1,, T2)
(TST, i, T1,, 2, L2)      // Branch to L2 if i<b+c
(MOV, 0,, T2)
(LBL, L2)
(TST, T2, 0,, 6, L3)      // If true, jump to loop body
(JMP, L4)                 // Exit loop
(LBL, L5)
(ADD, i, 1, i)
(JMP, L1)                 // Repeat loop
(LBL, L3)
(DIV, b, ='2', T3)        // Loop Body
(MOV, T3,, b)
(JMP, L5)                 // Jump to increment
(LBL, L4)
```

2. Repeat Problem 1 for the atom string resulting from the C++ statement:

```
if (a==(b-33)*2) a = (b-33)*2;
     else a = x+y;
```

```
(SUB, b, ='33', T1)
(MUL, T1, ='2', T2)
(MOV, 1,, T3)
(TST, a, T2,, 1, L1)      // If a == T2, jump to L1
(MOV, 0,, T3)
(LBL, L1)
(TST, T3, 0,, 1, L2)      // If T3 is false, jump to L2
(SUB, b, ='33', T4)
(MUL, T3, ='2', T5)
(MOV, T5,, a)
(JMP, L3)                 // Skip else part
(LBL, L2)
(ADD, x, y, T6)
(MOV, T6,, a)
(LBL, L3)
```

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, and Motorola 680x0 architectures each has sixteen general purpose registers, for example, and the RISC (Reduced Instruction Set Computer) architectures, such as the SUN SPARC and MIPS, 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 [1990].

An example, showing the importance of smart register allocation, is shown in Figure 6.7 for the two statement program segment:

```
A = B + C * D ;
B = 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 6.7 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

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	
MUL	R1,D			
STO	R1,Temp2			
LOD	R1,A			
SUB	R1,Temp2			
STO	R1,B			

Figure 6.7 Register Allocation, Simple and Smart, for a Two Statement Program

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 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 6.8 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, 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 6.8 we generate the code shown in Figure 6.9. 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.

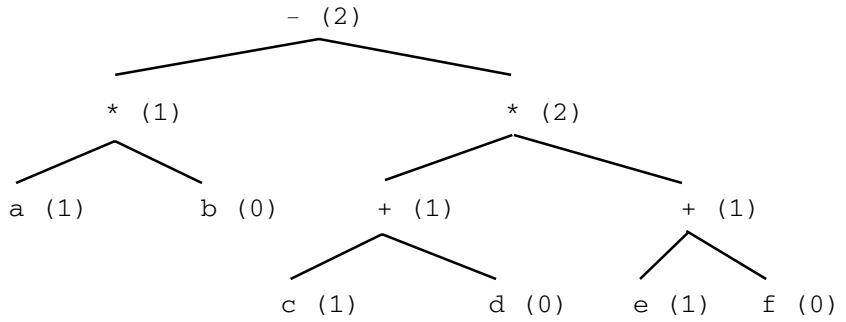


Figure 6.8 Weighted Syntax Tree for `a*b-(c+d)*(e+f)`, with Weights Shown in Parentheses

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.

```
LOD   R1,c
ADD   R1,d        R1 = c + d
LOD   R2,e
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,R1       R2 = a*b - (c+d)*(e+f
```

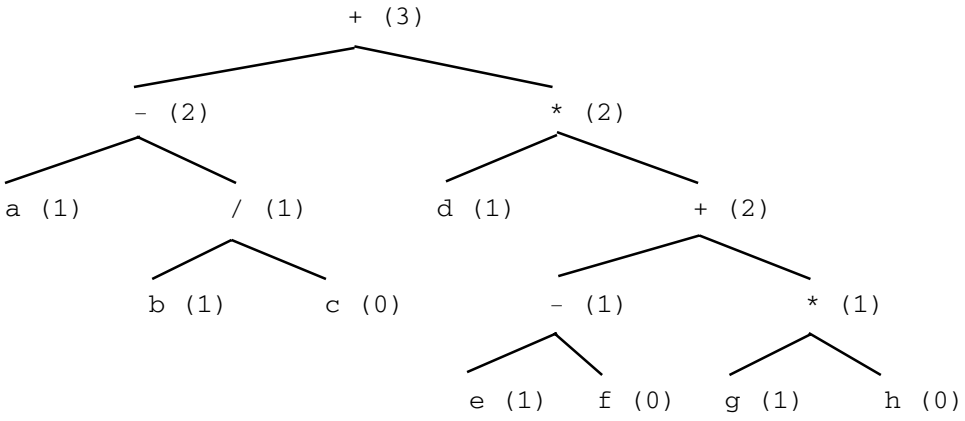
Figure 6.9 Code Generated for `a*b-(c+d)*(e+f)`, Using Figure 6.8

Sample Problem 6.4

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 6.9.

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)
```

Exercises 6.4

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 6.4. 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 MiniC 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 don't need to be concerned with efficiency or completeness. The architecture will be relatively simple and not cluttered with unnecessary features.

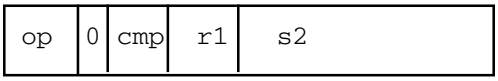
6.5.1 Mini: The Simulated Architecture

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

Absolute mode



Register-displacement mode

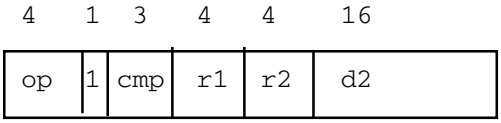


Figure 6.10 Mini Instruction Formats

(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 two instruction formats corresponding to the two addressing modes, as shown in Figure 6.10.

The ***absolute mode*** instruction can be described as:

```
fpreg[r1] ← fpreg[r1] op memory[s2]
```

and the ***register-displacement mode*** instruction can be described as

```
fpreg[r1] ← fpreg[r1] op memory[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
                                         → dest
10     NEG   left  —     result
                                         result ← - left
11     LBL   —     —     —     —         dest
                                         (no action)
12     TST   left  right -     cmp       dest
                                         → 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.4, 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

Multiply atom to compute `A*B`, putting result into `T1`:

class	left	right	result	cmp	dest
MUL	A	B	T1	—	—

```
gen (LOD, r = reg(), inp.left);
gen (MUL, r, inp.right);
gen (STO, r, inp.result);
```

Loc	Contents	
0		A
1		B
2	70100000	
3	30100001	
4	80100010	
...		
10		T1

Figure 6.11 Translation of a Multiply Atom

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.

Sample Problem 6.5

Show the code gerated by the code generator for the following TST atom. Assume that the value of `L1` is hex 23 and the variables `A` and `B` are stored at locations 0 and 1, respectively.

class	left	right	result	cmp	dest
TST	A	B	—	4	L1

Solution:

Loc	Contents			
0		A		
1		B		
2	70100000		LOD	R1,A
3	64100001		CMP	R1,B,4
4	50000023		JMP	L1

Exercises 6.5

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
MOV	T1	—	B	—	—
LBL	—	—	—	—	L2

(b)

class	left	right	result	cmp	dest
NEG	A	—	T1	—	—
LBL	—	—	—	—	L1
MOV	T1	—	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* for the MiniC language. 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.4, 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 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 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.

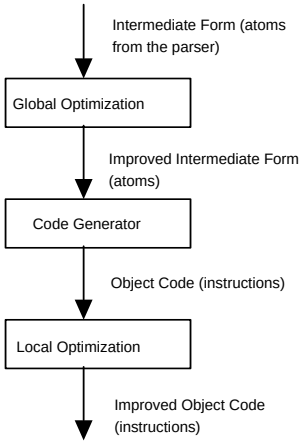


Figure 7.1 Sequence of Optimization Phases in a Compiler

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 don't 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.

Exercises 7.1

1. Using a C++ compiler,
(a) what would be printed as a result of running the following:

```
{
int a;
(a = 2) + (a = 3);
cout << 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 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 C++ statement:

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

The sequence of atoms put out by the parser could conceivably be as shown in Figure 7.2 below:

```
(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) ;`

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, below, 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.

```
(ADD, b, c, T1)
(MUL, T1, T1, a)
```

Figure 7.3 Optimized Atom Sequence for `a = (b + c) * (b + c) ;`

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 C++ statement, and the example is, admittedly, artificially contrived to make the point that Label atoms will affect our ability to optimize.)

```
(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

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.

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

`(ADD, b, c, T1)`	`Block 1`
`(LBL, L1)`	
`(ADD, b, c, T2)` `(MUL, T1, T2, T3)`	`Block 2`
`(TST, b, c,, 1, L3)`	
`(MOV, T3,, a)`	`Block 3`

Figure 7.5 Basic Blocks Contain No LBL, TST, or JMP Atoms

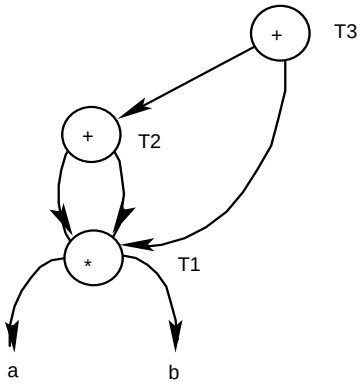


Figure 7.6 Example of a DAG

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, below, 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:

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)
```

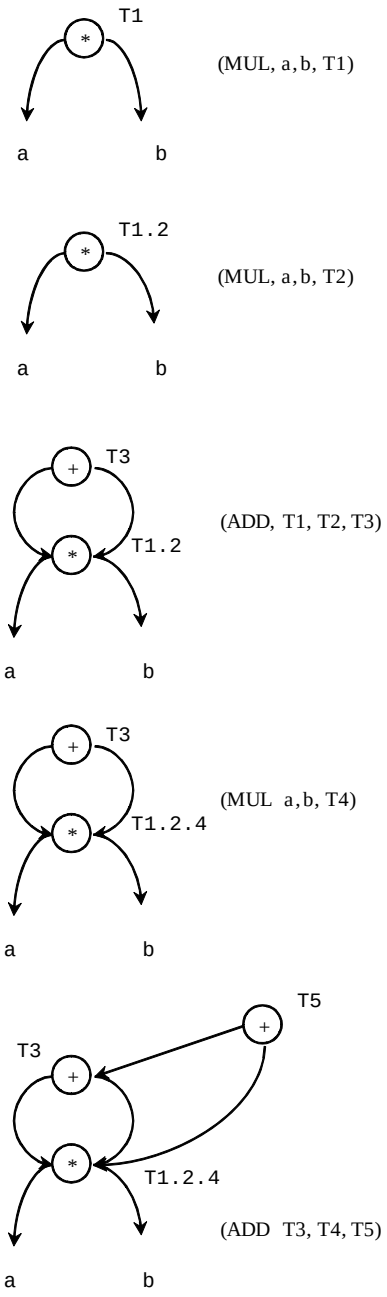


Figure 7.7 Building the DAG for `a * b + a * b + a * b`

```
(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`. 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.

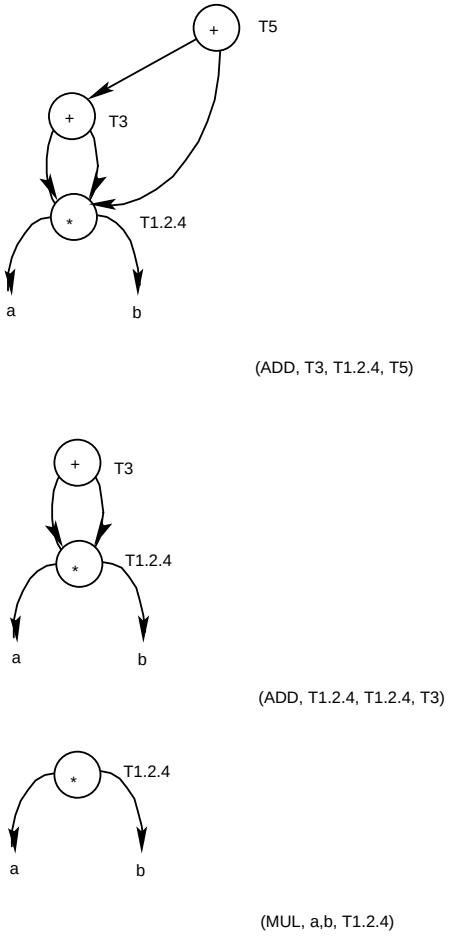


Figure 7.8 Generating Atoms from the DAG for `a * b + a * b + a * b`

This algorithm is demonstrated below, 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.

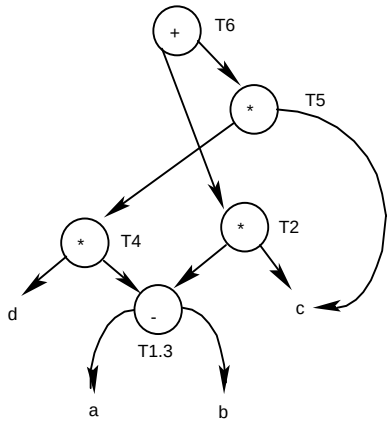
Sample Problem 7.2 (a)

Construct the DAG and show the optimized sequence of atoms for the C++ 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)
```

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)        ⇒        (JMP, L1)
(ADD, T2, d, T3)                 (LBL, L2)
(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. In general, a JMP atom should always be followed by an 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`.

```
{
       a = b + c * d;       //This statement has no effect and can be removed
       b = c * d / e;
       c = b - 3;
       a = b - c;
       cout << a << b << c ;
}
```

Figure 7.9 Elimination of Dead Code

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, below.

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

```
{
       for (i=0; i<1000; i++)
           { a = sqrt (x);              // loop invariant
             vector[i] = i * a;
           }
}

{      a = sqrt (x);              // loop invariant
       for (i=0; i<1000; i++)
           {
             vector[i] = i * a;
           }
}
```

Figure 7.10 Movement of Loop Invariant Code

```
{
        a = 2 * 3;                  // a must be 6
        b = c + a * a;              // a * a must be 36
}


{
        a = 6;
        b = c + 36;
}
```

Figure 7.11 Constant Folding

could be 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, above, 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 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 certain 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, below, in Figure 7.12. We do not believe that these properties

`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

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 (b)

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)
```

Exercises 7.2

1. Eliminate *common subexpressions* from each of the following strings of atoms, using DAGs as shown in Sample Problem 7.2 (a) (we also give the C++ 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 C++ functions, eliminate statements which constitute *dead code*. In each case, the function returns a value to the calling function in the parameter `d`:

(a)
```
void f (int & d)
{ int a,b,c;
     a = 3;
     b = 4;
     d = a * b + d;
}
```

(b)
```
void f (int & d)
{ int a,b,c;
     a = 3;
     b = 4;
```

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

5. In each of the following C++ 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 C++ program segments:

(a)
```
a = 2 + 3 * 8;
b = b + (a - 3);
```

(b)
```
void f (int & c)
{    const int a = 44;
     const int b = a - 12;
     c = a + b - 7;
}
```

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: Load the first operand into a register (LOD), perform the operation, and 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 C++ 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).

Sample Problem 7.3

Use the peephole methods of load/store, jump over jump, and simple algebraic optimization to improve the following Mini program segment:

```
      CMP   R1,a,2                  // JMP if R1 < a
      JMP   L1
      CMP   0,0,0
      JMP   L2
L1:
      LOD   R1,b
      ADD   R1,c
      STO   R1,T1
      LOD   R1,T1
      SUB   R1,a
      STO   R1,T2
      LOD   R1,T2
      STO   R1,a
      SUB   R1,0
      STO   R1,b
L2:
```

Solution:

```
      CMP   R1,a,2                  // Jump Over Jump
      JMP   L1
      CMP   0,0,0
      JMP   L2
```

```
L1:
        LOD   R1,b
        ADD   R1,c
        STO   R1,T1              // Load/Store
        LOD   R1,T1
        SUB   R1,a
        STO   R1,T2              // Load/Store
        LOD   R1,T2
        STO   R1,a
        SUB   R1,0               // Algebraic
        STO   R1,b
L2:


        // optimized code
        CMP   R1,a,5             // JMP if R1 >= a
        JMP   L2
        LOD   R1,b
        ADD   R1,c
        SUB   R1,a
        STO   R1,a
        STO   R1,b
L2:
```

Exercises 7.3

1. Optimize each of the following code segments for unnecessary *Load/Store* instructions:

(a)
```
LOD  R1,a
ADD  R1,b
STO  R1,T1
LOD  R1,T1
SUB  R1,c
STO  R1,T2
LOD  R1,T2
STO  R1,d
```

(b)
```
LOD  R1,a
LOD  R2,c
ADD  R1,b
ADD  R2,b
STO  R2,T1
ADD  R1,c
LOD  R2,T1
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) *consant 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.

Appendix A

MiniC Grammar

In this appendix, we give a description and grammar of the source language that we call "MiniC." ***MiniC*** is a simple subset of the standard C language. It does not include arrays, structs, unions, files, sets, switch statements, do statements, or or many of the low level operators. The only data types permitted are int and float. A complete grammar for MiniC is shown below, and it is similar to the yacc grammar used in the compiler in Appendix B.2. Here we use the convention that symbols beginning with upper-case 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.

```
Function    →    Type identifier ( ArgList ) CompoundStmt
ArgList     →    Arg
                 | ArgList , Arg
Arg         →    Type identifier
Declaration →    Type IdentList ;
Type        →    int
                 | float
IdentList   →    identifier , IdentList
                 identifier
Stmt        →    ForStmt
                 | WhileStmt
                 | Expr ;
                 | IfStmt
                 | CompoundStmt
                 | Declaration
                 | ;
ForStmt     →    for ( Expr ; OptExpr ; OptExpr ) Stmt
```

```
OptExpr      →     Expr
                   | ε
WhileStmt    →     while ( Expr ) Stmt

IfStmt       →     if ( Expr ) Stmt ElsePart
ElsePart     →     else Stmt
                   | ε
CompoundStmt→      { StmtList }
StmtList     →     StmtList  Stmt
                   | ε
Expr         →     identifier = Expr
                   | Rvalue
Rvalue       →     Rvalue Compare Mag
                   | Mag
Compare      →     == | < | > | <= | >= | !=
Mag          →     Mag + Term
                   | Mag - Term
                   | Term
Term         →     Term * Factor
                   | Term / Factor
                   | Factor
Factor       →     ( Expr )
                   | - Factor
                   | + Factor
                   | identifier
                   | number
```

This grammar is used in Appendix B.2 as the yacc grammar for our MiniC compiler, with very few 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.

MiniC is clearly a very limited programming language, yet despite its limitations it can be used to program some useful applications. For example, a MiniC program to compute the cosine function is shown in Figure A.1.

```
int main ()
{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 A MiniC Program to Compute the Cosine Function

Appendix B

MiniC Compiler

B.1 Software Files

The compiler for MiniC shown in this appendix is implemented using lex for the lexical analysis and yacc for the syntax analysis. The syntax phase 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 characters 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:

`mk`	Shell script to compile software using make utility
`makefile`	Input to make utility
`MiniC.l`	Input to lex
`MiniC.y`	Input to yacc
`MiniC.c`	Recursive descent parser
`gen.c`	Code generator
`mini.c`	Target machine simulator
`mini.h`	Header file for simulator
`MiniC.h`	Header file for compiler
`cos.c`	MiniC program to compute the cosine function
`bisect.c`	MiniC program to compute the square root of two by bisection
`fact.c`	MiniC program to compute the factorial function
`locate`	Shell script to find file containing specified source code
`readme`	Updated information regarding the software

The source files are available at:

`http://www.rowan.edu/~bergmann/books/miniC`

The unix/linux make command can be used to generate executables.

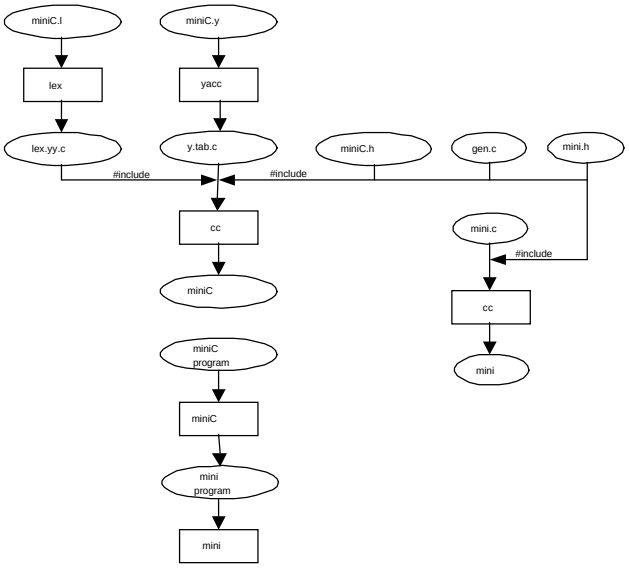


Figure B.1 Flow Diagram to Generate the Compiler and Simulator

After copying these files to a new directory, simply invoke the `make` command to compile and link the software. If you make any changes to the source code, again invoke `make` to compile only those modules which need to be compiled.

To compile the cosine program and view the output on the monitor, use the command:

```
$ MiniC <cos.c
```

To compile the cosine program and store the output in a file named `cos.mini`, use the command:

```
$ MiniC <cos.c >cos.mini
```

To run this program on the simulator, first compile the mini simulator:

```
$  cc mini.c -o mini
```

then run the cosine program using the command:

```
$ mini <cos.mini
```

To compile the cosine program, and execute it on the simulator with one command:

```
$ MiniC <cos.c | mini
```

A flow graph indicating the relationships of these files is shown in Figure B.1 in which input and output file names are shown in ovals and executing programs are shown in rectangles.

B.2 Lexical Phase

The lexical analysis phase is implemented with the Unix lex utility. The input to lex is the source file `MiniC.l`, and the output is the file `lex.yy.c`, which includes the `yylex()` function. This function reads MiniC source code from `stdin` (the standard input file) and returns tokens to the parser. The tokens are listed in the `MiniC.y` file (though single character tokens are not listed there). In addition to key words, the lexical phase finds comparison tokens, the assignment token, comments, numeric constants, and identifiers. White space and newline characters are taken as token delimiters, but ignored otherwise.

The function `searchIdent()` is used to install and look up identifiers in a hash table, which forms the symbol table for this compiler. The `searchIdent()` function checks for the use of undeclared identifiers and multiply-declared identifiers and puts out an appropriate error message.

The function `searchNums()` is used to install numeric constants into a binary search tree so that constants used more than once in the source program are stored only once in the run-time program. The value of the constant is stored as a floating-point number in the target machine's memory.

The function `alloc(size)` allocates the specified number of words of memory for use at run time. It is used to allocate space on the target machine for declared identifiers.

As explained in Appendix B.3 below, the parser stack may store three different types of data: a target machine address (`address`), an integer code (`code`), and a structure of three label numbers (`labels`). Consequently in our lex source file when we assign a value to the global variable `yylval`, we must indicate which of the three types is being returned, because it will be pushed onto the parser stack. For example, when a numeric constant is found, we assign its location in the target machine memory to `yylval: yylval.address = searchNums();`.

The source code forming the input to lex, taken from the file `MiniC.l`, is shown below:

```
INT   [0-9]+
EXP   ([eE][+-]?{INT})
NUM   ({INT}\.?{INT}?|\.{INT}){EXP}?
%{
#include <stdlib.h>
ADDRESS searchIdent(void);
ADDRESS searchNums(void);
ADDRESS alloc(int size);
%}
%start COMMENT1 COMMENT2
%%
<COMMENT1>.+      ;
<COMMENT1>\n      BEGIN 0;          /* end comment */
<COMMENT2>[^*]+   ;
<COMMENT2>\*[^/]  ;
<COMMENT2>\*\/    BEGIN 0;          /* end comment */
"//"         BEGIN COMMENT1;
"/*"         BEGIN COMMENT2;
int          {yylval.code = 2; return INT;}
float        {yylval.code = 3; return FLOAT;}
for          return FOR;
while        return WHILE;
if           return IF;
else         return ELSE;
"=="         {yylval.code = 1; return COMPARISON;}
\<           {yylval.code = 2; return COMPARISON;}
>            {yylval.code = 3; return COMPARISON;}
"<="         {yylval.code = 4; return COMPARISON;}
```

```
">="        {yylval.code = 5; return COMPARISON;}
"!="        {yylval.code = 6; return COMPARISON;}
[a-zA-Z][a-zA-Z0-9_]*   {yylval.address = searchIdent();
                         return IDENTIFIER;}
{NUM}       {yylval.address = searchNums(); return NUM;}
[ \t]       ;  /* white space */
\n          lineno++;         /* free format */
.           return yytext[0]; /* any other char */
%%
yywrap ()
{return 1;  /* terminate when reaching end of stdin */}

ADDRESS searchIdent(void)
/* search the hash table for the identifier in yytext.
insert if
necessary */
{ struct Ident * ptr;
  int h;
  h = hash(yytext);
  ptr = HashTable[h];
  while ((ptr!=NULL) && (strcmp(ptr->name,yytext)!=0))
      ptr = ptr->link;
  if (ptr==NULL)
     if (dcl)
      { ptr =  malloc (sizeof (struct Ident));
        ptr->link = HashTable[h];
        strcpy (ptr->name = malloc (yyleng+1), yytext);
        HashTable[h] = ptr;
        ptr->memloc = alloc(1);
        ptr->type = identType;
      }
     else { printf ("%s \n", yytext);
            yyerror ("undeclared identifier");
            return 0;
          }
  else if (dcl)
            { printf("%s \n", yytext);
              yyerror ("multiply defined identifier");
            }
  return ptr->memloc;
}

int hash(char * str)
{ int h=0, i;
  for (i=0; i<yyleng; i++) h += str[i];
```

```
  return h % HashMax;
}

ADDRESS searchNums(void)
/*  search the binary search tree of numbers for the number
in
    yytext.  Insert if not found. */
{ struct nums * ptr;
  struct nums * parent;
  double val;
  sscanf (yytext,"%lf",&val);
  if (numsBST==NULL)
   { numsBST = malloc (sizeof (struct nums));
     memory[numsBST->memloc = alloc(1)].data = val;
     numsBST->left = numsBST->right = NULL;
     return numsBST->memloc;
   }
  ptr = numsBST;
  while (ptr!=NULL)
   { if (memory[ptr->memloc].data==val) return ptr->memloc;
     parent = ptr;
     if (memory[ptr->memloc].data<val) ptr = ptr->left;
     else ptr = ptr->right;
   }
  ptr = malloc (sizeof (struct nums));
  memory[ptr->memloc = alloc(1)].data = val;
  ptr->left = ptr->right = NULL;
  if (memory[parent->memloc].data<val) parent->left = ptr;
    else parent->right = ptr;
  return ptr->memloc;
}

ADDRESS alloc (int size)
/* allocate words on the target machine memory */
{
  ADDRESS t;
  t = avail;
  avail += size;
  return t;
}
```

B.3 Syntax Analysis

The syntax analysis phase is implemented with the yacc utility and is stored in the file MiniC.y. This file contains the main program, which shows that control is initially given to the `yyparse()` function, which calls the `yylex()` function when it needs a token. The `yyparse()` function creates a file of atoms, and when there are no more tokens in the input stream, it calls the code generator. The file `MiniC.y` is shown below:

```
%{
#include <stdio.h>
#include "mini.h"
#include "miniC.h"
%}
%union {
      ADDRESS  address;
      int code;   /* comparison code 1-6 */
      struct {int L1;
              int L2;
              int L3;
              int L4;} labels;
       }
%token <address> IDENTIFIER
%token <code> INT
%token <code> FLOAT
%token FOR
%token WHILE
%token <code> COMPARISON
%token IF
%token ELSE
%token <address> NUM
/* nonterminal types are: */
%type <code> Type
%type <address> Expr
%type <address> OptExpr
%type <labels> WhileStmt
%type <labels> ForStmt
%type <labels> IfStmt
%type <labels> Label
%right '='
%left COMPARISON
%left '+' '-'
%left '*' '/'
%left UMINUS UPLUS
%%
```

```
Function:   Type IDENTIFIER '(' ArgListOpt ')' CompoundStmt
            ;
ArgListOpt: ArgList
            |
            ;
ArgList:    ArgList ',' Arg
            | Arg
            ;
Arg:        Type IDENTIFIER
            ;
Declaration: Type             {dcl = TRUE;
                               identType = $1;}
            IdentList ';'     {dcl = FALSE;}
            ;
IdentList:  IDENTIFIER ',' IdentList
            | IDENTIFIER
            ;
Type:       INT               {$$ = $1;}
            | FLOAT                 {$$ = $1;}
            ;
Stmt:       ForStmt
            | WhileStmt
            | Expr ';'
            | IfStmt
            | CompoundStmt
            | Declaration
            | ';'             /* null statement */
            ;
ForStmt:    FOR '(' Expr ';'  {$$.L1 = newlabel();
                         atom (LBL,NULL,NULL,NULL,0,$$.L1);}
            OptExpr ';'       {$$.L2 = newlabel();
                               atom (TST,$6,zero,NULL,6,
                                    $<labels>$.L2);
                               $$.L3 = newlabel();
                               atom (JMP,NULL,NULL,NULL,0,
                                    $<labels>$.L3);
                               $$.L4 = newlabel();
                               atom (LBL,NULL,NULL,NULL,0,
                                    $<labels>$.L4);}
            OptExpr ')'       {atom (JMP,NULL,NULL,NULL,0,
                                    $<labels>5.L1);
                               atom (LBL,NULL,NULL,NULL,0,
                                    $<labels>8.L2);}
             Stmt             {atom (JMP,NULL,NULL,NULL,0,
                                    $<labels>8.L4);
```

```
                                atom (LBL,NULL,NULL,NULL,0,
                                     $<labels>8.L3);}
            ;
OptExpr:    Expr                {$$ = $1;}
            |                   {$$ = one;} /* default to inf
loop */
            ;
WhileStmt:  WHILE               {$$.L1 = newlabel();
                                 atom(LBL,NULL,NULL,NULL,0,$$.L1);}
            '(' Expr ')'        {$$.L2 = newlabel();
                               atom (TST,$4, zero,
                                     NULL,1,$$.L2);}
            Stmt              {atom (JMP,NULL,NULL,NULL,0,
                                     $<labels>2.L1);
                               atom (LBL,NULL,NULL,NULL,0,
                                     $<labels>6.L2);}
            ;
IfStmt:           IF '(' Expr ')'   {$$.L1 = newlabel();
                   atom (TST, $3, zero, NULL, 1, $$.L1);}
            Stmt                {$$.L2 = newlabel();
                              atom (JMP,NULL,NULL,NULL,0, $$.L2);
                              atom (LBL,NULL,NULL,NULL,0,
                                     $<labels>5.L1);}
            ElsePart          {atom (LBL,NULL,NULL,NULL,0,
                                     $<labels>7.L2);}
            ;
ElsePart:
            | ELSE Stmt
            ;
CompoundStmt:     '{' StmtList '}'
            ;
StmtList:   StmtList Stmt
            |
            ;

Expr:       IDENTIFIER '=' Expr     {atom (MOV, $3, NULL,
                                     $1,0,0);
                                     $$ = $3;}
            | Expr COMPARISON Expr
              Label           {$$ = alloc(1);
                               atom (MOV, one, NULL, $$,0,0);
                               atom (TST, $1, $3, NULL, $2,
                                     $4.L1);
                               atom (MOV, zero, NULL, $$,0,0);
                              atom (LBL,NULL,NULL,NULL,0,$4.L1);}
```

```
            | '+' Expr %prec UPLUS  {$$ = $2;}
            | '-' Expr %prec UMINUS {$$ = alloc(1);
                                     atom (NEG, $2,NULL,$$,0,0); }
            | Expr '+' Expr   {$$ = alloc(1);
                               atom (ADD, $1, $3,$$,0,0); }
            | Expr '-' Expr   {$$ = alloc(1);
                               atom (SUB, $1, $3, $$,0,0); }
            | Expr '*' Expr   {$$ = alloc(1);
                               atom (MUL, $1, $3, $$,0,0); }
            | Expr '/' Expr   {$$ = alloc(1);
                               atom (DIV, $1, $3, $$,0,0); }
            | '(' Expr ')'    {$$ = $2;}
            | IDENTIFIER      {$$ = $1; }
            | NUM             {$$ = $1; }
            ;
Label:                        {$$.L1 = newlabel();}
            ;                 /* Used to store a label in
                                 compare expr above */
%%
char *progname;
char * op_text();
int lineno = 1;
ADDRESS save;
ADDRESS one;
ADDRESS zero;
int nextlabel = 1;
#include "lex.yy.c"
#include "gen.c"

main (int argc, char *argv[])
{
  progname = argv[0];
  atom_file_ptr = fopen ("atoms", "wb");
  strcpy (yytext,"0.0");
  zero = searchNums();  /* install the constant 0.0
                           in table  */
  strcpy (yytext, "1.0");
  one = searchNums();   /* also 1.0 */
  yyparse();
  fclose (atom_file_ptr);
  if (!err_flag) code_gen();
}

yyerror (char * s)
{
```

```
  fprintf(stderr, "%s[%d]: %s\n", progname, lineno, s);
printf ("yytext is <%s>", yytext);
  err_flag = TRUE;
}

newlabel (void)
{ return nextlabel++;}

/* testing only */  char mne[4];

atom (int operation, ADDRESS operand1, ADDRESS operand2,
                ADDRESS result, int comparison, int dest)
/* put out an atom. destination will be a label number. */
{ struct atom outp;

  outp.op = operation;
  outp.left = operand1;
  outp.right = operand2;
  outp.result = result;
  outp.cmp = comparison;
  outp.dest = dest;

  fwrite (&outp, sizeof (struct atom), 1, atom_file_ptr);
  /* testing only
  printf ("%d %x %x %x %d %d\n", operation, operand1,
           operand2, result, comparison, dest);
  decode (operation);
  printf ("  %s\n", mne);
  */
}

decode (int atom)
/*  Convert an atom number to a readable mnemonic */
{

  switch (atom)
    { case ADD:   strcpy (mne, "ADD");
            break;
      case SUB: strcpy (mne, "SUB");
            break;
      case MUL: strcpy (mne, "MUL");
            break;
      case DIV: strcpy (mne, "DIV");
            break;
      case JMP: strcpy (mne, "JMP");
```

```
            break;
      case NEG: strcpy (mne, "NEG");
            break;
      case LBL: strcpy (mne, "LBL");
            break;
      case TST: strcpy (mne, "TST");
            break;
      case MOV: strcpy (mne, "MOV");
    }
}
```

B.4 Code Generator

The code generator is written in the C language; the main function is named `code_gen()` and is stored in the file `gen.c`. There is a `#include` statement in the yacc program in `MiniC.y`, which incorporates the code generator into the MiniC compiler. The function `code_gen()` is called from the main program after the `yyparse()` function terminates. `Code_gen()` reads from the file of atoms and 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, above, in Appendix B.2.

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:

```
struct atom inp;
long labels[MAXL];
ADDRESS pc=0;
int ok = TRUE;

code_gen ()
{ int r;

  /* send target machine memory containing constants to
      stdout */
  end_data = alloc(0);              /* constants precede
                                       instructions */
  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);

  atom_file_ptr = fopen (“atoms”,”rb”); /* open file of
                                           atoms for */
  get_atom();                 /* second pass */
  pc = end_data;
  ok = TRUE;
 while (ok)
  {
 /*  dump_atom();  */

  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 its memory ad-
dress */
{ return labels[label_num];
}


out_mem()
/* send target machine memory contents to stdout.  this is
the beginning of the object file, to be followed by the
instructions.  the first word in the object file is the
starting address of the program; the next word is memory
location 0. */
{
  ADDRESS i;

  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);
}



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. The first two locations are reserved for the constants 0.0 and 1.0. For example, the program that computes the cosine function begins as shown here:

```
int main ()
{  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 regis-
ters 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
                                    even if stdin is
                                    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;      /* uncondi-
                          tional */
                          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 ad-
                        dress 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 minipas.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 `minipas.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];
```

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.

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.

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, $\hookleftarrow$, 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.

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.

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.

lex library A collection of run-time functions which may be called by the yylex() function.

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.

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.

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.

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.

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.

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.

start condition In the lex utility, a state which is entered by the scanner, resulting from a particular input; used to specify the left context for a pattern.

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's monitor.

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.

top down parsing Finding the structure of a string in a way that produces or traverses the derivation tree from top to bottom.

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.

yylex( ) The C function written by the lex utility to perform lexical analysis, using a pattern-matching mechanism.

yyparse( ) The C function, written by the yacc utility, to parse according to a given grammar.

Bibliography

Adams, J, et.al., *Turbo C++ An Introduction to Computing*, Upper Saddle River, NJ: Prentice Hall, 1996.

Aho, A. V. and J. D. Ullman *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing* Englewood Cliffs, NJ: Prentice Hall, 1972.

Aho, A. V. and J. D. Ullman *The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling* Englewood Cliffs, NJ: Prentice Hall, 1973.

Aho, A. V., R. Sethi, and J. D. Ullman *Compilers: Principles, Techniques, and Tools* Reading, MA: Addison Wesley, 1987.

Barrett, W. A., et. al. *Compiler Construction, Theory and Practice* Chicago: Science Research Associates, 1986.

Bennett, J. P. *Introduction to Compiling Techniques: A First Course using ANSI C, LEX, and YACC*, London: McGraw-Hill, 1990.

Berman, A.M., *Data Structures in C++*, New York: Oxford University Press, 1997.

Bornat, R. *Understanding and Writing Compilers* London: MacMillan, 1986.

Chomsky, N. *Syntactic Structures* The Hague: Mouton, 1965.

Chomsky, N., "Certain Formal Properties of Grammars" *Information and Control* Vol. 2, No. 2, June 1958: 137-167.

Cohen, D. I. A. *Introduction to Computer Theory* New York: Wiley, 1986.

Ellis, M., and B. Stroustrup *The Annotated C++ Reference Manual* Reading, MA: Addison Wesley, 1990.

Fischer, C. N., and R. J. LeBlanc *Crafting a Compiler with C* Menlo Park, CA: Benjamin-Cummings, 1991.

Ginsburg, S. *The Mathematical Theory of Context Free Languages* New York: McGraw-Hill, 1966.

Gries, D. *Compiler Construction for Digital Computers* New York: Wiley, 1968.

Holub, A. I. *Compiler Design in C* Englewood Cliffs, NJ: Prentice Hall, 1990.

Hopcroft, J. E., and J. D. Ullman *Introduction to Automata Theory, Languages, and Computation* Reading, MA: Addison Wesley, 1979.

Hutton, B., "Language Implementation," Unpublished manuscript, University of Auckland, NZ, 1987.

Johnson, S. C., "YACC - Yet Another Compiler Compiler," C. S. Technical Report No. 32, Murray Hill, NJ: Bell Telephone Labs, 1975.

Kamin, S. N. *Programming Languages: An Interpreter Based Approach* Reading, MA: Addison Wesley, 1990.

Kernighan, B. W., and R. Pike *The Unix Programming Environment* Englewood Cliffs, NJ: Prentice Hall, 1984.

Knuth, D. E., "On the Translation of Languages from Left to Right" *Information and Control* Vol. 8, No. 6, Dec. 1965: 607-639.

Lesk, M. E., and E. Schmidt, "LEX - a Lexical Analyzer Generator" In *Unix Programmer's Manual 2* Murray Hill: Bell Telephone Labs, 1975.

Lemone, K. A. *Fundamentals of Compilers: An Introduction to Computer Language Translation* Boca Raton, FL: CRC, 1992.

Lemone, K. A. *Design of Compilers: Techniques of Computer Language Translation* Boca Raton, FL: CRC, 1992.

Levine, J. R., et. al. *lex and yacc* Sebastopol, CA: O'Reilly, 1992.

Lewis, P. M., et. al. *Compiler Design Theory* Reading, MA: Addison Wesley, 1976.

Mak, R. *Writing Compilers and Interpreters* New York: Wiley, 1991.

Martin, S. *C Through Unix* Dubuque: Wm. C. Brown, 1992.

McKeeman, W. M., et. al. *A Compiler Generator* Englewood Cliffs, NJ: Prentice Hall, 1970.

Muchnick, S. S., *Advanced Compiler Design Implementation* San Francisco: Morgan Kaufmann, 1997.

Parsons, T. W., *Introduction to Compiler Construction*, New York: Freeman, 1992.

Pollack, B. W., ed. *Compiler Techniques* Princeton, NJ: Auerbach, 1972.

Pratt, T. W. *Programming Languages: Design and Implemenation* Englewood Cliffs, NJ: Prentice Hall, 1984.

Pyster, A. B. *Compiler Design and Construction* Boston: PWS, 1980.

Salomaa, A. *Formal Languages* New York: Academic Press, 1973.

Sethi, R. *Programming Languages: Concepts and Constructs* Reading, MA: Addison Wesley, 1989.

Schecter, P. B., and E. T. Desautels *An Introduction to Computer Architecture Using VAX Machine and Assembly Language* Dubuque: Wm. C. Brown, 1989.

Schreiner, A. T., and H. G. Friedman *Introduction to Compiler Construction with UNIX* Englewood Cliffs, NJ: Prentice Hall, 1985.

Stroustrup, B., *The Design and Evolution of C++* Reading, MA: Addison Wesley, 1994.

Tanenbaum, A. S. *Structured Computer Organization* Englewood Cliffs, NJ: Prentice Hall, 1990.

Tremblay, J. P., and P. G. Sorenson *The Theory and Practice of Compiler Writing* New York: McGraw-Hill, 1985.

Waite, W. H., and G. Goos *Compiler Construction* New York: Springer, 1984.

Waite, W. M., and L. R. Carter *An Introduction to Compiler Construction* New York: Harper Collins, 1993.

Wang, P. *An Introduction to Berkeley Unix* Belmont, CA: Wadsworth, 1988.

Wirth, N. *Compiler Construction*, Edinburgh: Addison Wesley Longman, 1996.

Index

Symbols

A

N

O

P

Q

R

S

T

U

V

W

Y