

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, $\{\text{boy, girl, animal}\}$ is a set of words, but it represents the same set as $\{\text{girl, boy, animal, girl}\}$. 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. $\{\epsilon, 0, 00, 000, 0000, 00000, \dots\}$
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. $\{\epsilon\}$
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

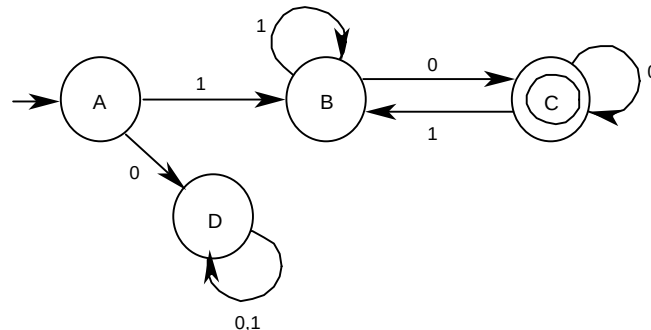


Figure 2.1 Example of a Finite State Machine

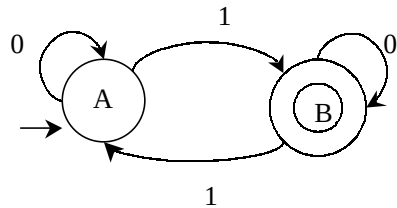


Figure 2.2 Even Parity Checker

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.

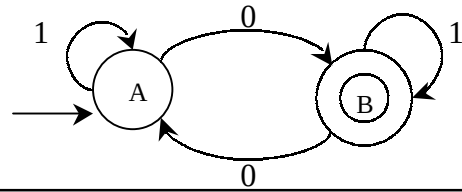
Sample Problem 2.0 (a)

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:

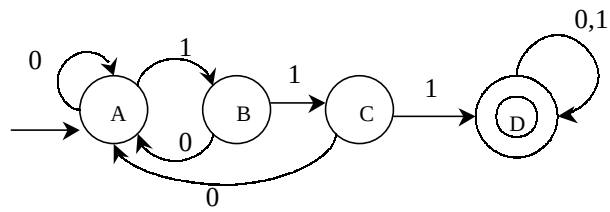
	0	1
A	B	A
*B	A	B



2. Strings containing three consecutive ones

Solution:

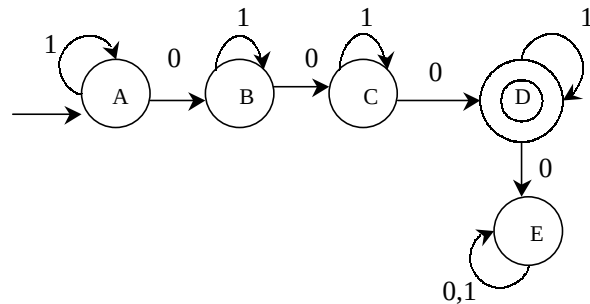
	0	1
A	A	B
B	A	C
C	A	D
*D	D	D



3. Strings containing exactly three zeros

Solution:

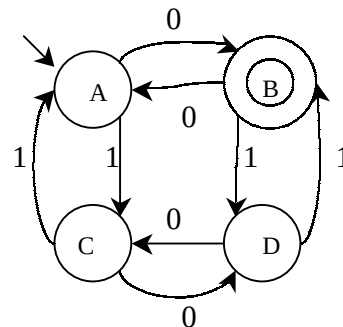
	0	1
A	B	A
B	C	B
C	D	C
*D	E	D
E	E	E



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,

$$abc \cdot ba = abcba$$

Note that any string concatenated with the null string is that string itself: $s \cdot \epsilon = s$. In what follows, we will omit the quote marks around strings to avoid cluttering the page needlessly. The concatenation of two languages is that language formed by concatenating each string in one language with each string in the other language. For example,

$$\begin{aligned} \{ab, a, c\} \cdot \{b, \epsilon\} &= \{ab \cdot b, ab \cdot \epsilon, a \cdot b, a \cdot \epsilon, c \cdot b, c \cdot \epsilon\} \\ &= \{abb, ab, a, cb, c\} \end{aligned}$$

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, $L \cdot \{\epsilon\} = L$, but $L \cdot \phi = \phi$.

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

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^1 &= L \\ L^2 &= L \cdot L \end{aligned}$$

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

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

Note that $\phi^* = \{\epsilon\}$. Intuitively, Kleene * generates zero or more concatenations of strings from the language to which it is applied. We will use a shorthand notation in regular expressions – if x is a character in the input alphabet, then $x = \{“x”\}$; i.e., the character x represents the set consisting of one string of length 1 consisting of the character x . This simplifies some of the regular expressions we will write:

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

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

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

$$L^0 = \{\epsilon\}$$

$$L^1 = \{0, 1\}$$

$$L^2 = L \cdot L^1 = \{00, 01, 10, 11\}$$

$$L^3 = L \cdot L^2 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

$$L^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, 0000, \dots\}$$

= the set of all strings of zeros and ones.

Another example:

$$1 \cdot (0+1)^* \cdot 0 = 1(0+1)^*0$$

$$= \{10, 100, 110, 1000, 1010, 1100, 1110, \dots\}$$

= the set of all strings of zeros and ones which begin with a 1 and end with a 0.

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

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

$$L + (L + L) = (L + L) + L$$

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

Sample Problem 2.0 (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 | abcbcd |
| 2. | $(a+b)^* \cdot (c+d)$ | c | d | ac | abd | babc | bad |
| 3. | $(a^*b^*)^*$ | ϵ | a | b | ab | ba | aa |
- Note that $(a^*b^*)^* = (a+b)^*$

Exercises 2.0

- Suppose L_1 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 L_1 ?

(a)	0101	(b)	110211	(c)	000
(d)	010011	(e)	ϵ		
- Suppose L_2 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 L_2 ?

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].

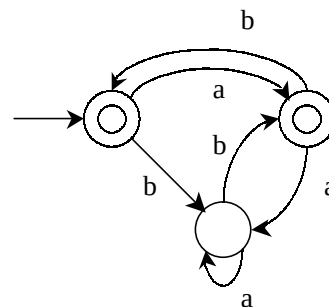
- (a) bca (b) accbab (c) ϵ
 (d) aaa (e) aabbcc

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 your own words, the *language* specified by each of the following finite state machines with alphabet $\{a, b\}$.

(a)

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

(b)

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

(c)

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

(d)

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

(e)

	a	b
A	B	B
*B	B	B

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

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

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

9. Construct *finite state machines* which specify the same language as each of the following regular expressions.

- (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.

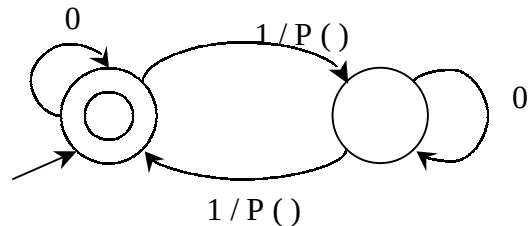
A finite state machine which accepts numeric constants is shown in Figure 2.5. Note that these constants must begin with a digit, and numbers such as .099 are not acceptable. This is the case in some languages, such as Pascal, whereas C++ does permit constants which do not begin with a digit. We could have included constants which begin with a decimal point, but this would have required additional states.

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 array 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

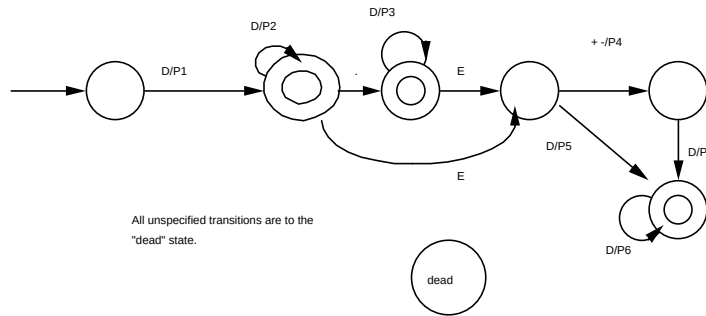
Sample Problem 2.2

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:

$\text{Result} = N * \text{Power}(10, \text{Sign} * \text{Exp} - \text{Places});$

where $\text{Power}(x, y) = x^y$

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()
{
    sum = L;
}
```

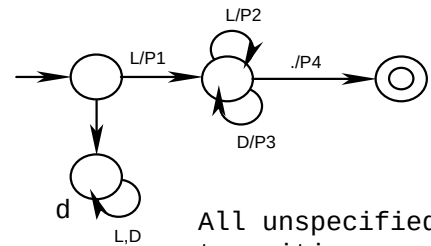
```
Void P3()
{
    sum += D;
}
```

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

```
Void P2()
{
    sum += L;
}
```

```
int Hash (int n)
{
    return n % 10;
}
```

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



All unspecified transitions are to state d.

6. Show the *values* that will be assigned 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

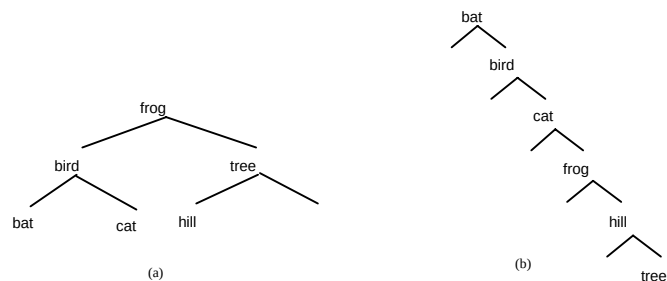


Figure 2.8 (a) A Balanced Binary Search Tree (b) A Binary Search Tree Which is Not Balanced

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 $\text{hash}(\text{bird}) = (4+98) \% \text{HASHMAX}$ where HASHMAX is the size of the array of pointers. The resulting value will always be in the range $0 \dots \text{HASHMAX} - 1$ 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

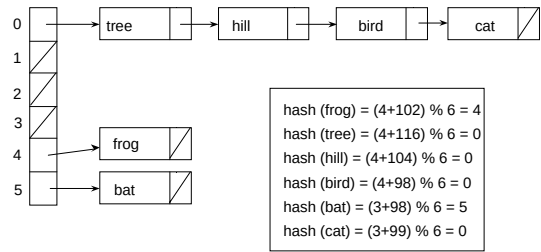


Figure 2.9 Hash Table Corresponding to the Words Entered for Figure 2.8(a)

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 alphabetical order (worst case)
 - (c) $2^n - 1$ identifiers, perfectly balanced
 - (d) n identifiers, 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 `yyllex()` 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 `yyllex()` 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.

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

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

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 `yyllex()` function as a subprogram rather than as a main program. The `yyllex()` 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 `yyllex()` each time it needs a token. Consequently, our `lex` program (which is used to generate the `yyllex()` 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 processing 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 `yyval.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 `yyval.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. Its purpose is to permit house-keeping 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.

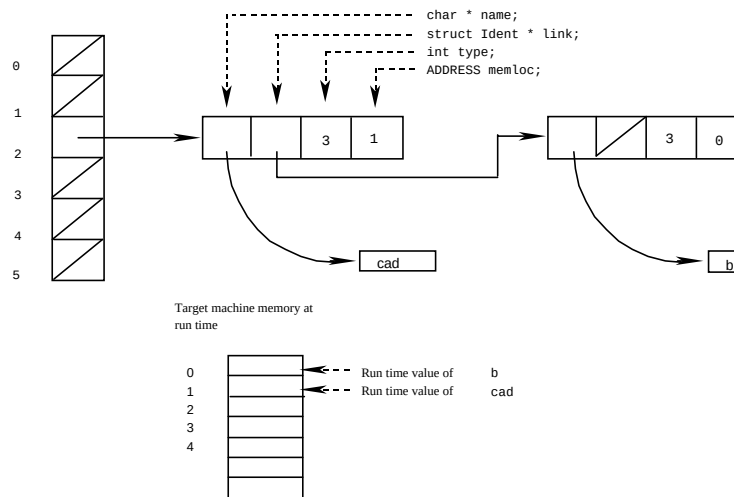


Figure 2.10 A Hash Table Storing the Identifiers `cad` and `b`, with Target Machine Memory

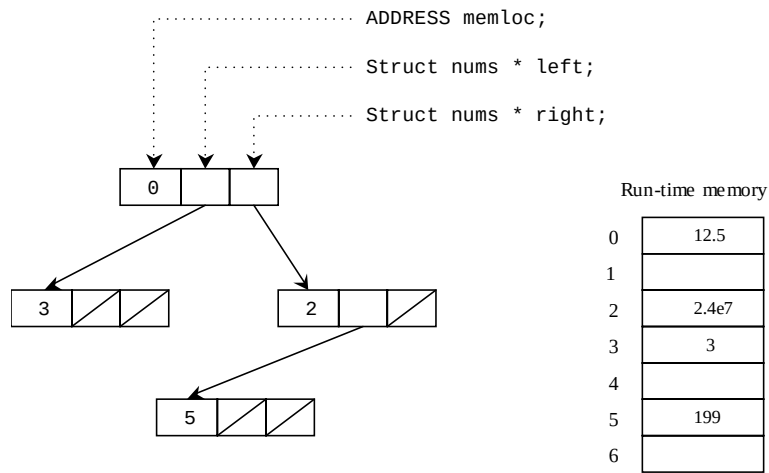


Figure 2.11 Binary Search Tree Storing the Constants 12.5, 2.4e7, 3, and 199, With Target Machine Memory

The `searchNums()` supporting function is very similar to `searchIdent()`. 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:

<u>Input</u>	<u>Output</u>
"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.