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. Java syntax
4. Italian syntax

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

2.0.2 Finite State Machines

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

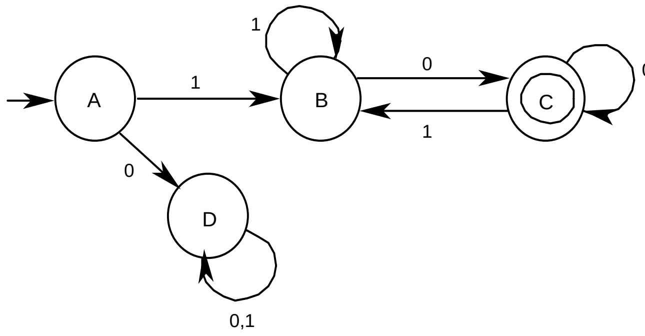


Figure 2.1 Example of a Finite State Machine

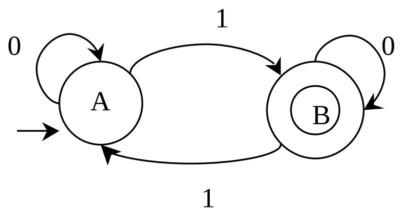


Figure 2.2 Even Parity Checker

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

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

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

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

		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.

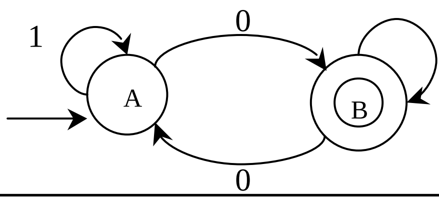
Sample Problem 2.0 (a)

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:

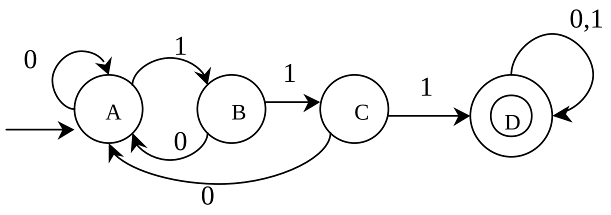
	0	1
A	B	A
*B	A	B



2. Strings containing three consecutive ones

Solution:

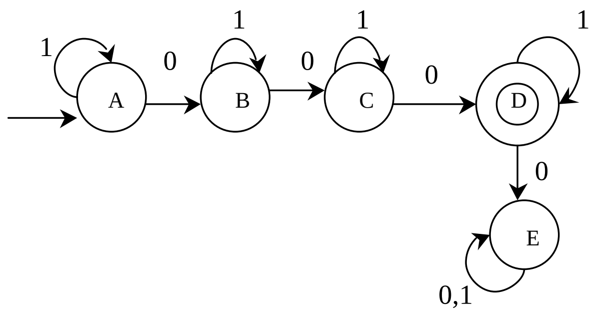
	0	1
A	A	B
B	A	C
C	A	D
*D	D	D



3. Strings containing exactly three zeros

Solution:

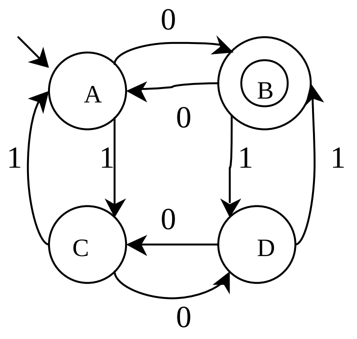
	0	1
A	B	A
B	C	B
C	D	C
*D	E	D
E	E	E



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: $s \cdot \varepsilon = s$. 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,

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

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, $L \cdot \{\varepsilon\} = L$, 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:

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

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

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

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:

$1 \cdot (0+1)^* \cdot 0 = 1(0+1)^*0$
$= \{10, 100, 110, 1000, 1010, 1100, 1110, \ldots\}$
= 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?

(a) bca (b) accbab (c) ε

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].

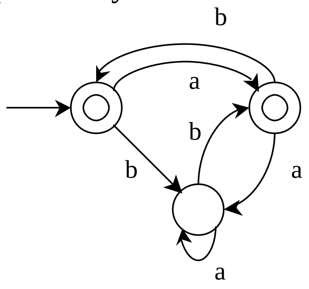
(d) `aaa` (e) `aabbcc`

3. Which of the following are examples of languages?

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

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

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



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

6. Describe, in 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 (ε is excluded from this language).

2.1 Lexical Tokens

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

(1) ***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 Java source input, showing the word boundaries and types is given below:

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

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

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

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

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

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

Token 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. Java is case sensitive.

Exercises 2.1

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

(a)
```
for (i=start; i<=fin+3.5e6; i=i*3)
        ac=ac+/*incr*/1;
```

(b)
```
{ ax=33;bx=/*if*/31.4 } // ax + 3;
```

(c)
```
if/*if*/a)}+whiles
```

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

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

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

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

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

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

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

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

2.2 Implementation with Finite State Machines

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

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

```
boolean [] accept = new boolean [STATES];
int [][] fsm = new int[STATES][INPUTS]; // state table
// initialize table here...
int inp = 0;                    // input symbol (0..INPUTS)
int state = 0;                  // starting state;
try
{  inp = System.in.read() - '0'; // character input,
                              // convert to int.

   while (inp>=0 && inp<INPUTS)
   {  state = fsm[state][inp];      // next state
      inp = System.in.read() - '0'; // get next input
   }
} catch (IOException ioe)
   {  System.out.println ("IO error " + ioe); }

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

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

2.2.1 Examples of Finite State Machines for Lexical Analysis

An example of a finite state machine which accepts any identifier beginning with a letter and followed by any number of letters and digits is shown in Figure 2.4. The letter "`L`" represents any letter `(a-z)`, and the letter "`D`" represents any numeric digit `(0-9)`.

This implies that a preprocessor would be needed to convert input characters to tokens suitable for input to the finite state machine.

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

A third example of the use of state machines in lexical analysis is

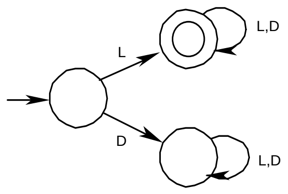


Figure 2.4 Finite State Machine to Accept Identifiers

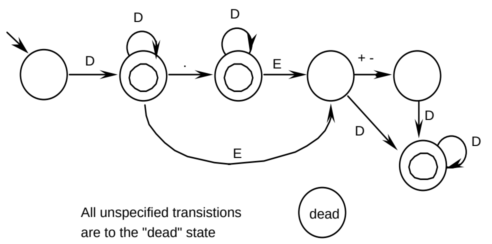


Figure 2.5 A Finite State Machine to Accept Numeric Constants

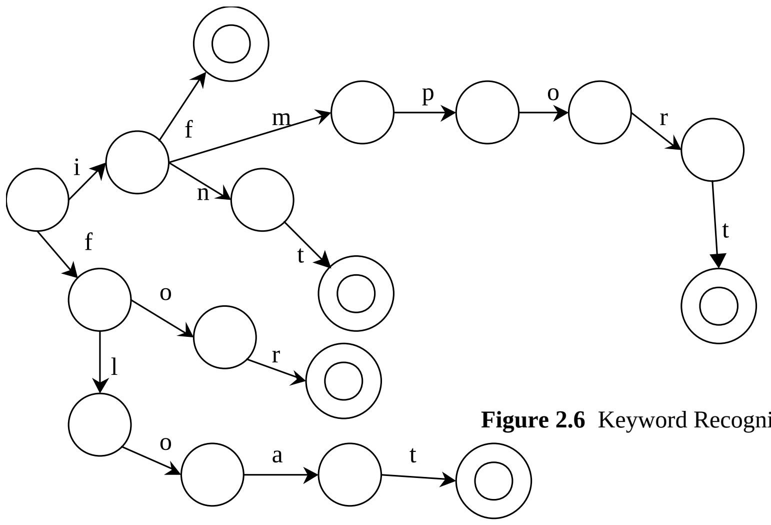


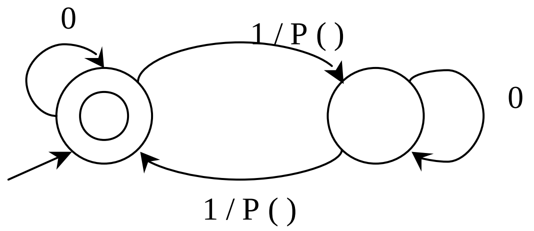
Figure 2.6 Keyword Recognizer

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

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



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

Figure 2.7 Parity Bit Generator

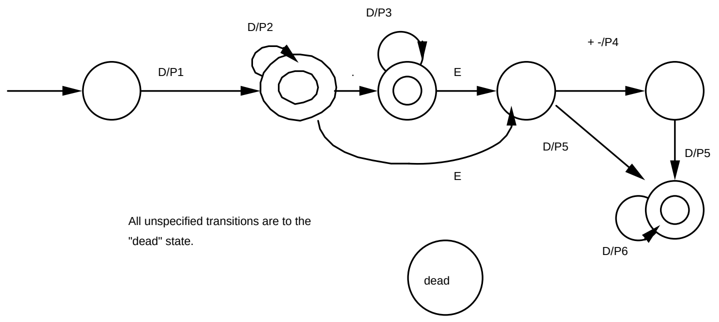
Sample Problem 2.2

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 * Math.pow (10, Sign*Exp  - Places);
```

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

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, above, 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()`.

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 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.

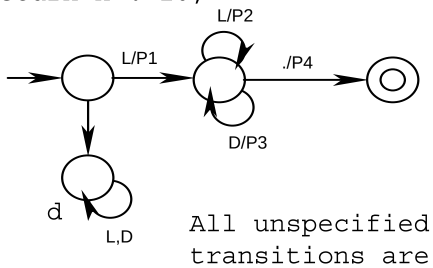
5. What is the *output* of the finite state machine, below, for each of the following inputs (`L` represents any letter, and `D` represents any numeric digit; also, assume that each input is terminated with a period):

```
int sum;

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

void P3()                  int hash (int n)
{                          {
   sum += D;                  return n % 10;
}                          }

Void P4()
{
System.out.println(hash(sum));
}
```



```
All unspecified
transitions are
to  state d.
```

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

6. Show the *values* that will be asigned to the variable `N` in Sample Problem 2.2 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

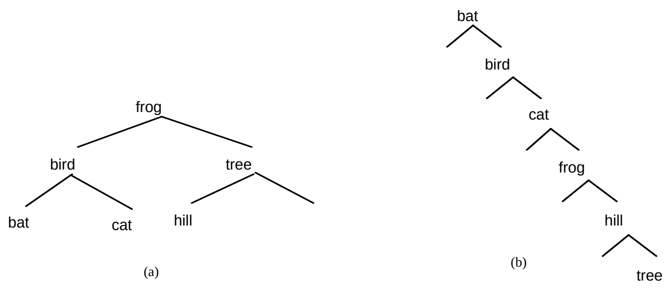


Figure 2.8 (a) A Balanced Binary Search Tree (b) A Binary Search Tree Which is Not Balanced

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

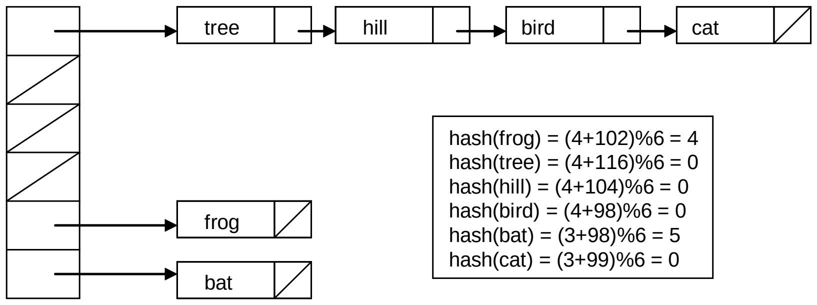

tree
hill
bird
cat
frog
bat
hash(frog) = (4+102)%6 = 4
hash(tree) = (4+116)%6 = 0
hash(hill) = (4+104)%6 = 0
hash(bird) = (4+98)%6 = 0
hash(bat) = (3+98)%6 = 5
hash(cat) = (3+99)%6 = 0

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

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

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

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

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

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

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

(a) `ab, ac, ad, ae`
(b) `ae, bd, cc, db`
(c) `aa, ba, ca, da`

7. Show a sequence of four *identifiers* which would cause your hash function in Problem 6 to generate a collision for each identifier after the first.

2.4 Lexical Analysis with SableCC

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

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

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

Unlike the lex/yacc utilities which could be used separately or together for lexical and syntax analysis, respectively, SableCC combines lexical and syntax analysis into a single program. Since we have not yet discussed syntax analysis, and we wish to run SableCC for lexical analysis, we provide a SableCC template for the student to use.

2.4.1 SableCC Input File

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

1 Package declaration
2 Helper declarations
3 States declarations
4 Token declarations
5 Ignored tokens
6 Productions

At the present time the student may ignore sections 5 and 6, whereas the sections on Helper declarations, States declarations, and Token declarations are relevant to lexical analysis. The required Java classes will be provided to the student as a standard template. Consequently, the input file, named *language*.grammar will be arranged as shown below:

Package *package-name* ;

Helpers
    [Helper declarations, if any, go here]

States
    [State declarations, if any, go here]

Tokens
    [Token declarations go here]

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

2.4.1.1 Token Declarations

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

Token-name = Token-definition ;

For example: `left_paren = '(' ;`

A Token definition may be any of the following:
A character in single quotes, such as 'w', '9', or '$'.
A number, written in decimal or hexadecimal, representing the ascii (actually unicode) code for a character. Thus, the number 13 represents a newline character (the character '\n' works as well).
A set of characters, specified in one of the following ways:
A single quoted character qualifies as a set consisting of one character.
A range of characters, with the first and last placed in brackets:

```
['a'..'z']  // all lower case letters
['0'..'9']  // all numeric characters
[9..99]     // all characters whose codes are in
            // the range 9 through 99,inclusive
```

A union of two sets, specified in brackets with a plus as in [set1 + set2].
Example:

```
[['a'..'z'] + ['A'..'Z']]      // represents all
                               // letters
```

A difference of two sets, specified in brackets with a minus as in [set1 - set2].
This represents all the characters in set1 which are not also in set2. Example:

```
[[0..127] - ['\t' + '\n']]
            // represents all ascii characters
            // except tab and newline.
```

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

Regular expressions, with some extensions to the operators described in section 2.0, may also be used in token definitions. If p and q are token definitions, then so are:

`(p)`	parentheses may be used to determine the order of operations (precedence is as defined in section 2.0).
`pq`	the concatenation of two token definitions is a valid token definition.
`p|q`	the union of two token definitions (note the plus symbol has a different meaning).
`p*`	the closure (kleene *) is a valid token definition, representing 0 or more repetitions of p.
`p+`	similar to closure, represents 1 or more repetitions of the definition p.
`p?`	represents an optional p, i.e. 0 or 1 repetitions of the definition p.

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

```
number = ['0'..'9']+ ;          // A number is 1 or more
                                // decimal digits
identifier = [['a'..'z']+[['A'..'Z']]
              (['a'..'z'] | ['A..'Z'] | ['0'..'9'] | '_')* ;
                  // An identifier must begin with an
                  // alphabetic character.
rel_op = ['<' + '>'] '='? | '==' | '!=' ;
                  // Six relational operators
```

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

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

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

Instead the tokens should be defined as:

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

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

2.4.1.2 Helper Declarations

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

```
Helpers
  digit = ['0'..'9'] ;
  letter = [['a'..'z'] + ['A'..'Z']] ;
  sign = '+' | '-' ;
  newline = 10 | 13 ;         // ascii codes
  tab = 9 ;                   // ascii code for tab
Tokens
  number = sign? digit+ ;     // A number is an optional
                              // sign, followed by 1 or more
                              // digits.

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

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

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

Sample Problem 2.4(a)

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

```
334 abc abc334
```

Solution:

```
number           334
space
identifier       abc
space
identifier       abc334
```

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

2.4.1.3 State Declarations, Left Context, and Right Context

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

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

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

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

The first state listed is the start state; the scanner will start out in this state.

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

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

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

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

A definition may be associated with more than one state:

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

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

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

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

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

```
                           // Scanning a newline char.  Apply this in
                           // either state.  New state is bol.
```

Sample Problem 2.4 (b)

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

Solution:

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

States
  def, currency;              // def is start state.

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

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

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

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

In the text:

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

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

2.4.1.4 An Example of a SableCC Input File

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

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

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

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

2.4.2 Running SableCC

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

```
package lexing;
```

```
import lexing.lexer.*;
import lexing.node.*;
import java.io.*;        // Needed for pushbackreader and
                         // inputstream
class Lexing
{
static Lexer lexer;
static Object token;

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

There is now a two-step process to generate your scanner. The first step is to generate the Java class definitions by running SableCC. This will produce a sub-directory, with the same name as the language being compiled. All the generated java code is placed in this sub-directory. Invoke SableCC as shown below:

```
sablecc languagename.grammar
```

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

```
sablecc lexing.grammar
```

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

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

In our case this would be:

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

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

```
java languagename.Classname
```

In our case this would be:

```
java lexing.Lexing
```

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

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

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

Exercises 2.4

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

(1) Identifier (begins with letter, followed by letters, digits, _)
(2) Numeric constant (float or int)
(3) = (assignment)
(4) Comparison operator (`==   <   >   <=   >=   !=`)
(5) Arithmetic operator (`+   -   *   /`)
(6) String constant `"inside double-quote marks"`
(7) Keyword (`if   else   while   do   for   class`)
Comments

```
/* Using this method */
// or this method, but don't print a token
//   class.
```

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

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

Input files:

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

2.5 Case Study: Lexical Analysis for Decaf

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

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

The Helpers section, shown below, defines a few macros which will be useful in the Tokens section. A letter is defined to be any single letter, upper or lower case. A digit is any single numeric digit. A digits is a string of one or more digits. An `exp` is used for the exponent part of a numeric constant, such as 1.34e12. A `newline` is an end-of-line character (for various systems). A `non_star` is any unicode character which is not an asterisk. A `non_slash` is any unicode character which is not a (forward) slash. A `non_star_slash` is any unicode character except for asterisk or slash. The helpers `non_star` and `non_slash` are used in the description of comments. The Helpers section, with an example for each Helper, is shown below:

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

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

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

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

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

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

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

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

  space = ' ' | 9 | newline ;    // 9 = tab
  clas = 'class' ;               // key words (reserved)
  public = 'public' ;
  static = 'static' ;
  void = 'void' ;
  main = 'main' ;
  string = 'String' ;
  int = 'int' ;
  float = 'float' ;
  for = 'for' ;
  while = 'while' ;
  if = 'if' ;
  else = 'else' ;
  assign = '=' ;
  compare = '==' | '<' | '>' | '<=' | '>=' | '!=' ;
  plus = '+' ;
  minus = '-' ;
  mult = '*' ;
  div = '/' ;
  l_par = '(' ;
  r_par = ')' ;
  l_brace = '{' ;
  r_brace = '}' ;
```

```
l_bracket = '[' ;
r_bracket = ']' ;
comma = ',' ;
semi = ';' ;
identifier = letter (letter | digit | '_')* ;
number  =  (digits '.'? digits? | '.'digits) exp? ;
misc = [0..0xffff] ;
```

This completes the description of the lexical analysis of Decaf. The implementation makes use of the Java class Hashtable to implement a symbol table and a table of numeric constants. This will be discussed further in chapter 5 when we define the Translation class to be used with SableCC.

Exercises 2.5

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

```
Input                      Hint
"A long string"            One string token
" Another 'c' string"      One string token
"one" 'x' "three"          A string, a char, a string
"  //   string "           A string, no comment
//  A "comment"            A comment, no string
```

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

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

DoStmt   →   do Stmt while ( Expr )
```

Show the necessary changes to the tokens section only.

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

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

2.6 Chapter Summary

Chapter 2, on ***Lexical Analysis***, began with some introductory theory of formal languages and automata. A *language*, defined as a set of strings, is a vital concept in the study of programming languages and compilers. An *automaton* is a theoretic machine, introduced in this chapter with *finite state machines*. It was shown how these theoretic machines can be used to specify programming language elements such as *identifiers*, *constants*, and *keywords*. We also introduced the concept of *regular expressions*, which can be used to specify the same language elements. Regular expressions are useful not only in lexical analysis, but also in utility programs and editors such as *awk*, *ed*, and *grep*, in which it is necessary to specify search patterns.

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

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

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

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