2

LEXICAL ANALYZER

The function of the lexical analyzer is to read the source program, one character at a time, and to translate it into a sequence of primitive units called *tokens* such as keywords, identifiers, constants and operators. A token is a sequence of characters that can be treated as a unit in the grammar of a programming language. This chapter discusses the different problems faced during the designing and implementation of lexical analyzers, such as representing tokens using regular expressions. A regular expression is a notation used to represent tokens of any programming language.

The regular expression can be expressed as transition diagrams, because those finite automations are convenient ways of designing token recognizers. One advantage of using regular expression to specify tokens is that, from a regular expression we can automatically construct a recognizer for tokens denoted by that regular expression.

2.1 INTRODUCTION TO LEXICAL ANALYZERS

In compilers, the lexical analyzer acts as a subroutine or function, which is called by the parser whenever it needs a new token. The lexical analyzer represents valid tokens as integer, and once it is found, returns to the parser.

In most of the compilers, the lexical analyzer and the parser are in the same pass. But in some compilers, they are in different passes to specify the structure of tokens more efficiently.

Other functions sometimes performed by a lexical analyzer are keeping track of line numbers, producing an output listing if necessary, skipping out white space, and deleting commands.

2.2 SCANNING THE INPUT

The lexical analyzer scans the characters of the source program one at a time to discover tokens; however, many characters beyond the next token may have to be examined before the next token itself can be determined. For this reason two pointers are used, one pointer to mark the beginning of the token being discovered, and the other, a look ahead pointer, to scan ahead of the beginning point, until the token is discovered. Figure 2.1 shows the how to scan the input tokens using look ahead pointer.

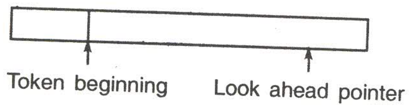

FIGURE 2.1 Scanning the Input.

2.3 DESIGN OF LEXICAL ANALYZERS

It is always wise and easy to begin the design of any program by describing the behaviour of the program by a flow chart. This approach is particularly very useful when the program is complex such as a lexical analyzer. It discovers tokens by scanning the characters from the input. We use special kind of flow charts named *transition diagrams* to represent the different operations of lexical analyzers. In a transition diagram, the boxes of the flow chart are drawn as circles called *state.* The states are connected by arrows called *edges.* The labels on the various edges leaving a state indicate the input characters that can appear after that state.

Figure 2.2 shows a transition diagram for a valid identifier or variable, defined to be a letter followed by any number of letters or digits. The starting state of the transition diagram is state S_0, called *initial state,* the edge from this state indicates that the first input character must be a letter. If this is the case, we enter state 1 and look for the next input character. If it happens to be a letter or digit, we re-enter state 1 and go for the next input character. We continue this process of reading letters and digits, and making transitions from state 1 to itself, until the next input character is a delimiter for an identifier (i.e. not a letter or a digit). On reading the delimiter, we enter into state S_2, which is also a *final state.* If any special characters are found from any states, they will go to the error state to show the appropriate error message.

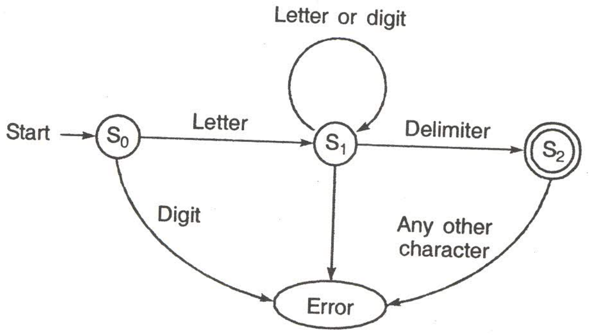

FIGURE 2.2 Transition Diagram for an Identifier.

Now we can convert these collections of transition diagrams into programs. Figure 2.3 shows the typical transition diagram of keywords used in the C language.

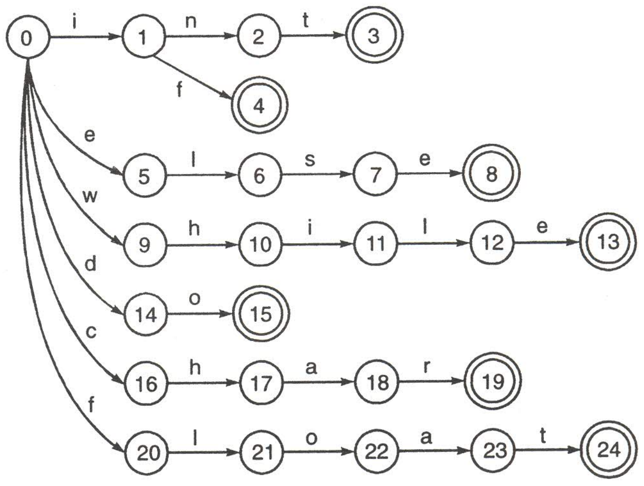

FIGURE 2.3 Transition Diagram for Keywords.

2.4 LANGUAGES

Let us first discuss few basic terms dealing with languages. The term "language" means a set of strings formed from some specific alphabet. We shall use the term alphabet to denote any finite set of symbols or character, which is the basic entity of a language.

For example, the set {0, 1} is an alphabet. It consists of two symbols, 0 and 1 and it is known as binary alphabets. A string is a finite sequence of symbols, such as 0011. The length of a string is the total number of symbols in it. For example, the length of a string 0011 is 4. A special string whose length is zero is called an *empty string*. The empty strings are denoted by ε, read as epsilon.

Let L and M be two languages, then the concatenation of two languages, $L.M$ or just LM, is the language consisting of all strings ab, where a is a string in language L and b is in M. That is,

$$LM = \{ab; \text{ where } a \text{ is in } L \text{ and } b \text{ is in } M\}$$

For example, let L {0, 10, 11} and M = {11, 1} be a language. Then LM = {011, 01, 1011, 101, 1111, 111}. Hence the string in LM 1011 can be written as the concatenation of 10 from L and 11 from M.

The empty set is a language, which contains only the empty string. It can be denoted as ϕ or $\{\varepsilon\}$.

If an empty set is concatenated with a language L, then we get the same language. That is,

$$\{\varepsilon\} L = L \{\varepsilon\} = L$$

The unit of two languages L and M is given by the set of all strings that contains either in language L or in M. That is,

$$L \cup M = \{x;\ \text{where } x \text{ is in } L \text{ or } x \text{ is in } M\}$$

$$L \cup M = \{\varepsilon\} = \{\varepsilon\} \cup L = L$$

2.5 REGULAR EXPRESSIONS

The regular expression (RE) is a very useful notation suitable for describing tokens, when the lexical analyzer is generated. Actually these regular expressions are converted automatically into finite automata which are just formal specifications of transition diagrams. So, in this section we will discuss how a regular expression can be used to represent a token.

Consider an example of identifiers in a programming language. An identifier is defined to be a letter followed by zero or more letters or digits. It can be represented as a regular expression as

Identifier = letter (letter/digit)*

The vertical bar means *or*, the parentheses are used to group sub expressions, and the start(*) is the closure operator meaning "zero or more instances".

In other words, an identifier can be defined as tokens that must have at least one character, which should be a letter, and followed by zero or more letter(s) or digits. Each of these regular expressions denotes a language, and the rules for constructing a regular expression that denotes a language are shown below.

1. ε is a regular expression denoting the language $\{\varepsilon\}$, and then the language will accept only the empty string.
2. If a is a regular expression denoting the language $\{a\}$, then the language will accept only one string, which is a.
3. If L and M are regular expressions denoting the language L_L and L_M, then:
   (a) $(L)/(M)$ is a regular expression denoting the language L_L or L_M.
   (b) $(L) \cdot (M)$ is a regular expression denoting the language L_L and L_M.
   (c) $(L)^*$ is a regular expression denoting the language $L_L{}^*$.

The * operator has the highest precedence followed by /. If two regular expressions L and M denote the same language, that is, $L = M$, then L and M are said to be equivalent languages. For example, $a^* = (a^*)^*$.

2.6 CLOSURE

A closure is an operator to show "any number of strings". We use the operator * for representing closure L^* to denote the concatenation of language L with itself any number of times, that is

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

For example, if $L = \{aa\}$, then the L^* means all strings of an even number of a's including empty string, that is, $L^0 = \{\varepsilon\}$, $L^1 = \{aa\}$, $L^2 = \{aaaa\}$, and so on. In other words, the language will accept all the strings that contain null string and an even number of a's.

If we want to exclude empty string, $\{\varepsilon\}$ we can denote the language as $L.(L^*)$, that is,

$$\begin{aligned} L \cdot (L^*) &= L \bigcup_{i=0}^{\infty} L^i \\ &= \bigcup_{i=0}^{\infty} L^{i+1} \\ &= \bigcup_{i=1}^{\infty} L^i \end{aligned}$$

In short, we can denote the language $L \cdot (L^*)$ as L^+. We use the unary postfix operator + to denote "one or more instance" of a string, and it is called *positive closure*.

For example, if $L = \{aa\}$, then the L^+ means all strings of an even number of a's excluding empty string. That is, $L^1 = \{aa\}$, $L^2 = \{aaaa\}$, and so on. Note that there is no L^0. Thus, the language will accept all the strings that contain only an even number of a's.

We can represent the regular expression of an identifier as follows.

```
Identifier = letter (letter/digit)*
Letter = A/B/C/D/...................../x/y/z
Digit = 0/1/2/3/......................../8/9
```

2.7 REGULAR EXPRESSIONS IN LEX

Before we describe the structure of the Lex specifications, let us discuss the regular expressions used by Lex. Regular expressions are widely used in rules section of the Lex specifications, which will be discussed in the next chapter.

In Lex, the regular expression is a pattern description using a meta language, a language that you use to describe particular patterns of interest. The characters used in this meta language are part of the standard ASCII character list. The characters that form regular expressions are:

.	matches any single ASCII character except the new line character ("\n").
*	matches zero or more copies of the preceding expression or character.
[ ]	This is to represent the character class that matches any character within the brackets.
^	If any character set or expression comes just after the operator, it will accept all the characters except the one within the character class, that is, this character can be used to negate any character class within the square brackets.
–	A dash inside the square brackets indicates a character range, for example [0–9] means the same thing as [0123456789] and [A–Z] means all the English alphabet letters A to Z.
{ }	It indicates how many times the previous pattern is allowed to match which contains one or two numbers. For example, A {1, 3} matches one to three occurrences of the letter A.
\	It is used to escape meta-characters and as part of the usual C escape sequence, for example, "\n" is a new line character, while "*" is literal asterisk.
+	Matches one or more characters of the preceding regular expressions or a character set. For example, [0–9] + matches one or more combinations of the character 0 to 9. That is, this regular expression accepts the strings 0112, 1231456, or 9012, but not an empty string.
?	It matches zero or one occurrence of the preceding regular expression, for example, –[0–9] + matches a signed number including an optional leading minus.
|	Matches any one of the preceding or following regular expressions, for example, pen/pencil/eraser. This expression will match any one of the three words.
"..."	It interprets everything within the equation marks as meta-characters other than the C escape sequences.
()	Groups a series of regular expressions together into a new expression. For example, (10101) represent a character sequence 10101.
<>	A name or a list of names in angle brackets at the beginning of a pattern makes that pattern apply only in the given start states.
<<EOF>>	In Flex, the special pattern matches the end of the file.

Usually, complex regular expressions are built up from these simple regular expressions by combining one or more.

2.8 EXAMPLES FOR REGULAR EXPRESSIONS

First, let us take the example that we had discussed earlier for an identifier.

```
identifier = letter (letter/digit)*
```

An identifier can be defined as the tokens that must have at least one character or letter and followed by zero or more letters or digits.

```
letter = [A-Z a-z]
digit = [0-9]
```

A *letter* can be defined as the any one character ranging from capital *A* to Z or small letter *a* to z and a digit as numerical *digits* ranging from 0 to 9. This can be used to build a regular expression for an integer.

```
integer = [0-9]+
```

An integer can be any one digit followed by zero or more digits. We can expand this to allow decimal numbers.

```
[0-9]*\.[0-9]+
```

Notice that the use of "\" before the period is to make it a literal period rather than a meta-character. That is, simply "." means, it matches any single character except the new line character, so we used "\." to specify the period of the decimal numbers. This pattern matches "0.0", "1.2" or ".312". But it will not match "0" or "2".

Following are the few examples for the regular expressions discussed above:

a	matches the character '*a*'.
[*abc*]	A character class that matches the pattern, either *a*, *b* or *c*.
[*abi–uZ*]	A character class that matches "a", "b" any letter from "*i*" through "*u*" and a "Z".
[^*A–Z*\n]	A negated character class, i.e. any character except an upper case letter or a new line.
*a**	Zero or more *a*'s where *a* is any regular expression pattern.
a⁺	one or more *a*'s.
a?	Zero or one *a*.
a{2, 5}	Two to five number of *a*'s.
a{3}	Three or more *a*'s.
a{4}	exactly four *a*'s.
\0	A NULL character (ASCII code 0).
\123	The character with octal value 123.
\x2*a*	The character with hexadecimal value 2*a*.
a$	An *a* but only at the end of a line, i.e. just before a new line equivalent to "*r*/\n".

There are several other regular expressions that can be specified from the combinations of the pattern that we have already discussed. Those regular expressions will be dealt while discussing programming with Flex.

2.9 FINITE AUTOMATA

The lexical analyzer can be viewed as a recognizer to identify the tokens from the input strings. The lexical analyzer that identifies the presence of a token on the input, is a recognizer for the language, using the regular expression defining that token.

A better way to convert a regular expression to a recognizer is to construct a generalized transition diagram from the expression. This diagram is called a Nondeterministic Finite Automata (NFA). In general, it can be converted into its variant, which is a simpler program, called a Deterministic Finite Automata (DFA). A NFA recognizing the language *(a/b)*abb* is shown in Figure 2.4.

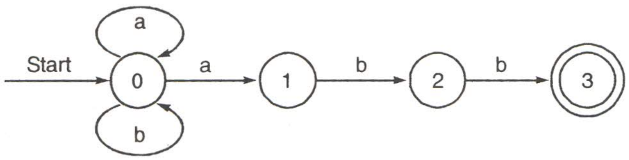

FIGURE 2.4 NFA Accepting *(a/b)*abb*.

If we analyze Figure 2.4 closely, we can see that the nondeterministic finite automata is a labelled directed graph. The values are called *states* and the labelled edges are called *transitions*. Here state 0 is a start (or initial) state, and state 3, with double circles, is a final state. The NFA has two or more transition states by accepting a character. Figure 2.4 shows a typical NFS outputting the language *(a/b)*abb*.

The DFA has only one transition state by accepting a character. Figure 2.5 shows the DFA of Figure 2.4.

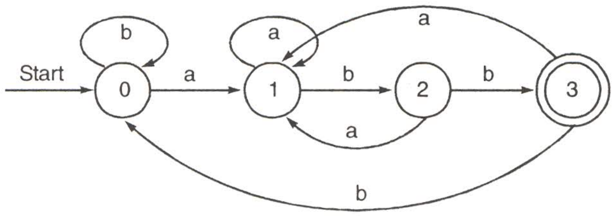

FIGURE 2.5 DFA Accepting *(a/b)*abb*.

Since there is at most one transition from any state for any input symbol, a DFA is easier to simulate than an NFA. Fortunately, for each NFA we can find a DFA accepting the same language. You may refer any standard textbook related to "Theory of Computation & Automata" for more details on finite automata.