Q.28. Explain Arden's theorem.

Ans. Let P and Q be two regular expressions over Σ. If P does not contain $\in$, then the following equation in R, Viz.

$$R = Q + RP \quad \text{....(i)}$$

has a unique solution (i.e., one and only one solution) given by

$$R = QP^*$$

Proof –

$$Q + (QP^*)P = Q(\in + P^*P) = QP^* \text{ by } I_9$$

Hence, equation (i) is satisfied when $R = QP^*$. This means $R = QP^*$ is a solution of (i).

To prove uniqueness, consider (i). Here, replacing R by Q + RP on the R.H.S., we get the equation

$$\begin{aligned} Q + RP &= Q + (Q + RP)P \\ &= Q + QP + RPP \\ &= Q + QP + RP^2 \\ &= Q + QP + QP^2 + \ldots\ldots + QP^i + RP^{i+1} \\ &= Q + (\in + P + P^2 + \ldots\ldots + P^i + P^{i+1}) + RP^{i+1} \end{aligned}$$

From (i),

$$R = Q(\in + P + P^2 + \ldots\ldots + P^i) + RP^{i+1} \text{ for } i \geq 0$$

Q.29. Explain Lexical analyzer. What is the role of the lexical analyzer in compiler ?

Ans. The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. As shown in fig. 1.20 upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

Since, the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. Some tasks that are performed by lexical analyzer are stripping out from source program comments and white space in the form of blank, tab and newline characters. Another is correlating error message from the compiler with the source program. In some compilers, lexical analyzer makes a copy of the source program with the error message marked in it. If source language supports some macro preprocessor junctions, then these preprocessor junctions may also be implemented during lexical analysis.

Sometimes, lexical analyzers are divided into a cascade of two phases, the first called *"scanning"* and the second *"lexical analysis"*. The scanner is responsible for doing simple tasks, while lexical analyzer does more complex operations.

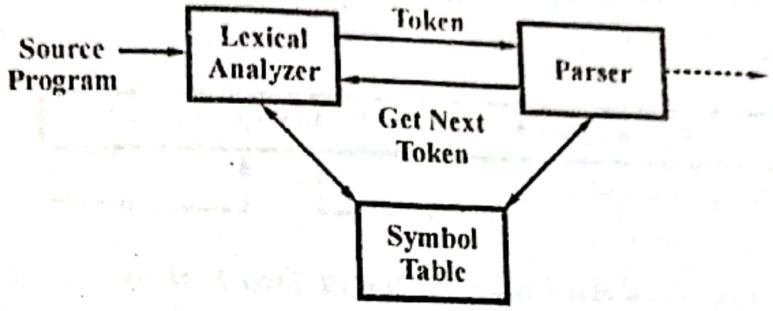


Fig. 1.20 Interaction of Lexical Analyzer with Parser

The interface of a lexical analyzer can be shown in fig.1.21. It reads characters from the input, groups them into lexemes and passes the tokens formed by the lexemes, together with their attributes values, to the later stages of compilers. In some cases, lexical analyzer has to read some characters ahead before it can decide on the token to be returned to the parser. The extra character has to be pushed back onto the input, because, it can be the beginning of next lexeme in the input.

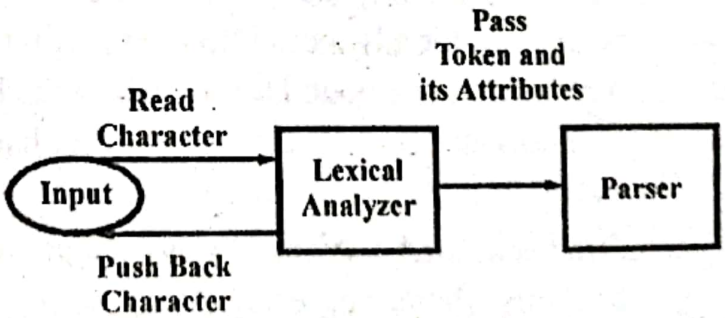


Fig. 1.21 Inserting a Lexical Analyzer between the Input and the Parser

Lexical analyzer build some useful meaning word and considers them as ***TOKEN*** which are defined as sequence of characters having a collective meaning. These tokens are then passed by analyzer to the next phase called ***syntax analyzer.*** There are variety of tokens each have certain information related bits called ***ATTRIBUTES.***

Depending upon the types of token each type thus have a specific patter For a given grammer token types are fixed for a valid program. Output of lexical analyzer called ***lexeme.*** The lexical analyzer produces tokens and th parser consumes them & form a producer-consumer pair.

Q.30. Explain input buffering briefly. ***(R.G.P.V., Dec. 2006)***

Ans. Input buffering is done for increasing efficiency of the compiler. A two-buffer input scheme is useful when look ahead on the input is necessary to identify tokens. Sentinels are used to mark the buffer end. This technique is useful for speeding up the lexical analyzer.

The input to the lexical analyzer is the string of characters of facilitates character, an input buffer is kept. A buffer divided into two N-character halves as shown is fig.1.22.

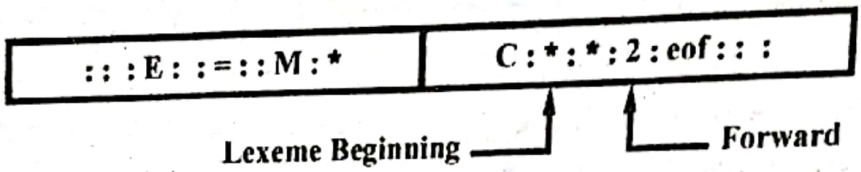


Fig. 1.22 An Input Buffer in Two Halves

Typically, N is the number of characters on one disk block e.g.– 1024 or 4096. Two pointers to the input buffer are maintained. The string of characters between the two pointer is the current lexeme. Initially, both pointers point to the first character of the next lexeme to be found. One of pointer is kept fixed at a lexeme beginning and other one is moved from left to right, called ***forward pointer,*** scans ahead until a match for a pattern is found. This lexeme is then passed to the outside buffer. Next, the fixed pointer is dragged to the next lexeme beginning and the process repeats. During the movement of the forward pointer if a boundary comes. Then automatically the other empty half is filled with next set of characters and pointer advances into next half whenever right block get secanned the pointer sensing a boundary returns to the beginning of left block and vice versa the forward pointer can also retract back wherever pushback function is evolved.

The amount of lookahead is limited with this buffering scheme, this make it impossible to recognize tokens where the distance that the forward pointer must travel is more than the length of the buffer.

Q.31. Describe the role of a lexical analyzer and also explain the concept of input buffering. ***(R.G.P.V., June 2009)***

Ans. **Role of a Lexical Analyzer** – Refer to Q.29.

Input Buffering – Refer to Q.30.

Q.32. Give the reasons for the separation of scanner and parser in separate phases of compiler. *(R.G.P.V., Dec. 2005)*

Or

What are various issuses in lexical analyzer ?

Ans. There are several reasons for separating the analysis phase of compiling into scanners (lexical analysis) and parsing (syntax analysis).

(i) Simpler Design – Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis allows us to simplify one or other of these phases. For example a parser embodying the conventions for comments and white space is more complex than one that can assume comments and white space have already been removed by a lexical analyzer.

(ii) Clean Overall Language Design – If we are designing a new language, separating the lexical (scanner) and parsing conventions can lead to a cleaner overall language design.

(iii) Compiler Efficiency – A separate lexical analyzer allows us to construct a specialized and potentially more efficient processor for the task. Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler. Thus the compiler efficiency is improved.

(iv) Compiler Portability – Portability is enhanced. Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer. The representation of special or non standard symbols can be isolated in lexical analyzer e.g.– symbol ↑ in pascal.

Q.33. Write down function of the lexical analyzer.

Ans. Various functions performed by lexical analyzer includes the following –

(i) Removal of White Space – The white space constitutes of blanks, tabs and newlines etc that appears between the tokens. They are detected and dropped by lexical analyzer so that they never appear at the output.

(ii) Removal of Comments – Comments can likewise be ignored by the parser and translator. So these too are filtered.

(iii) Handling of Constants – Anytime a single digit appears in an expression, it allows an arbitrary integer constant in its place. An integer constant is a sequence of digits, integer constants can be allowed either by adding productions to the grammer for expressions, or by creating a token for such constants. The job of collecting digits into integer to detect the size of sequence and then declare it as token "num" is done by lexical analyzer. The value of this constant is passed as an attribute with the token "num". The lexical analyzer passes both the token and the attributes to the parser. combination is

expressed as a tuple enclosed between <and> for e.g. – a constant 54 will appear at the output as

<num, 54>

(iv) Handling Operators – Similar action is taken this time generating the tuple such as <+, > or simply LE., GT etc without any attribute.

(v) Recognition of Identifiers and Keywords – Identifier are the name of variables, arrays, functions etc. They all appear under the token name "**identifier**" or "**id**". Simply for example –

count = count + increment ;

would be converted by lexical analyzer into token stream, used for parsing.

id = id + id;

Since every such identifier will have separate identify another table get created side by side which is called **symbol table**. In this table there is a entry for every identifier in first column of any row. The rest of the column carries other valuable information related to the id.

Keywords are fixed character strings such as begin, end, if etc, satisfy the rules for forming identifiers. Keywords are always reserved. Keyword of a language also finds an entry in table but all such keywords are passed by their name only.

Q.34. What are error recovery actions to be taken to recover from lexical errors in lexical analysis phase ?

Ans. A lexical analyzer gets a very localized view of source program. If any string 'fi' is encountered then the analyzer cannot tell it is a misspelled if and will simply pass it as an identifier.

In a situation when lexical analyzer is unable to proceed as because none of the tokens matches a prefix of the remaining input then its opts for a panic mode recovery. Where successive characters are deleted from the remaining input until and well formed token is detected. In an interactive computing enviroment such a technique is quite adequate.

Other possible error-recovery actions are –

(i) Deleting an extraneous character

(ii) Inserting a missing character

(iii) Replacing an incorrect character by a correct character.

(iv) Transposing two adjacent characters.

Error transformation like these may be tried in an attempt to repair the input. The strategy is to see whether a prefix of the remaining input can be transformed into a valid lexeme by just a single error transformation. One way of finding the errors in a program is to compute the minimum number of

error transformations required to transform the errorneous program into one which is syntactically correct.

However the lexical analyzer can only partially detect the errors the following phases of the compiler also implement many error detecting and recovery routine.

Q.35. How lexical analyzer recognize the token ?

Or

Explain how tokens are recognized. ***(R.G.P.V., June 2006)***

Ans. The recognition of token is done to separate out different tokens. To understand the concept let us consider that we have certain regular definition such as

if → if
then → then
else → else
relop → <|< = | = | < > | > | > =
id → letter (letter/digit)*
num → digit$^+$ (.digit$^+$) ? (E(+/-) ? digit$^+$) ?

Beside the above the appearances of white space can also be represented through regular definition as

delim → blank/tab/newline
ws → delim $^+$

Firstly the analyzer has to decide what kind of token is to be generated with what attribute for the incoming input string. A table as shown table 1.4 helps in this decision. The column attribute value if kept blank indicates that no attribute is to be passed otherwise the one which is indicated is to be passed for those definition such as relop or id or num where there are number of lexemes then to decide for the right one the help of transition diagrams are taken. For every definition there is a automata inside and thus a transition diagram.

Table 1.4 Regular Expression Patterns for Token

Regular Expression	Token	Attribute Value
WS	—	—
IF	IF	—
then	then	—
else	else	—
id	id	pointer to table entry
num	num	pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

A **Transition diagram** depicts the action that takes place when a lexical analyzer is called upon by a syntax analyzer to pass the next token.

(i) Recognition of Relational Operators – A model transition diagram to decide for the relational operator is shown in fig. 1.23. Starting from the start state (marked 0) which is the initial state, we proceed on any path after reading the next input character and finally landup in a final state with which is associated a function that passes the token with its attributes. There are certain final states which are marked with a star. These states have two functions associated, one is that of normal token passing and the other of pushbacking an extra character back to input buffer for e.g. – after reading a '<' operator another character has to be read to decide between a < or a <= if the equality sign appears as the next character then the token is passed for <= otherwise the token for < is passed and the extra character read is pushed back.

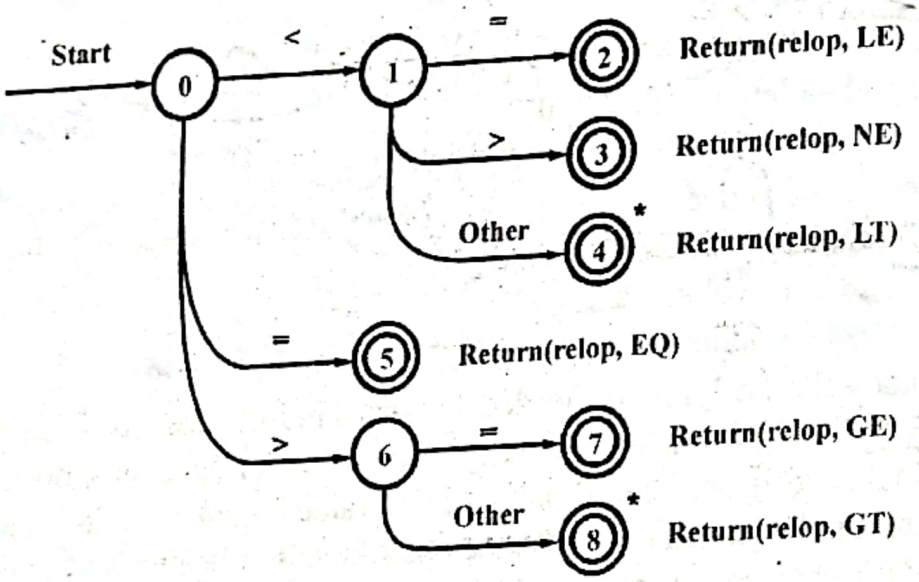


Fig. 1.23 Transition Diagram for Relational Operators

The transition diagram shown in fig. 1.24 is deterministic i.e., no symbol can match the label of two edges leaving one state. Also there will be several transition diagram each specifying a group of tokens. If failure occurs where we are following one transition diagram then we move to the start state and follow another transition diagram. To do so forward pointer is retracted in the input buffer to the lexeme beginning. If failure occurs for all transition digram then a lexical error is declared and error recovery routine are called.

(ii) Recognition of Identifiers and Keywords – As per the regular definition the transition diagram is as shown in fig. 1.24.

Here at the final state two functions get evoked beside the pushback character function. The function gettoken ( ) searches for the identifier detected in the symbol table. If it finds the same then its references are passed as

attribute with the token id otherwise another function install id is evoked which first enters the identifier name in the symbol table and then pass its references.

In case following the same pattern a keyword get detected then the corresponding action is taken by passing it by name or by its references in the keyword table which is another symbol table.

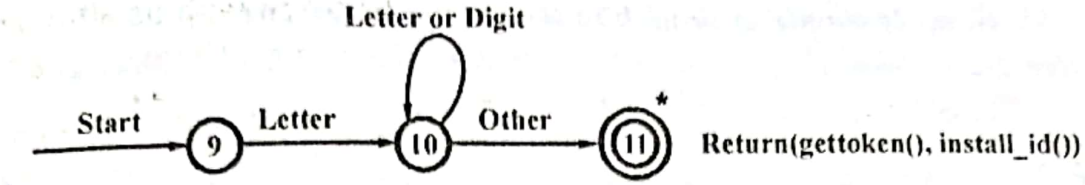


Fig. 1.24 Transition Diagram for Identifiers and Keywords

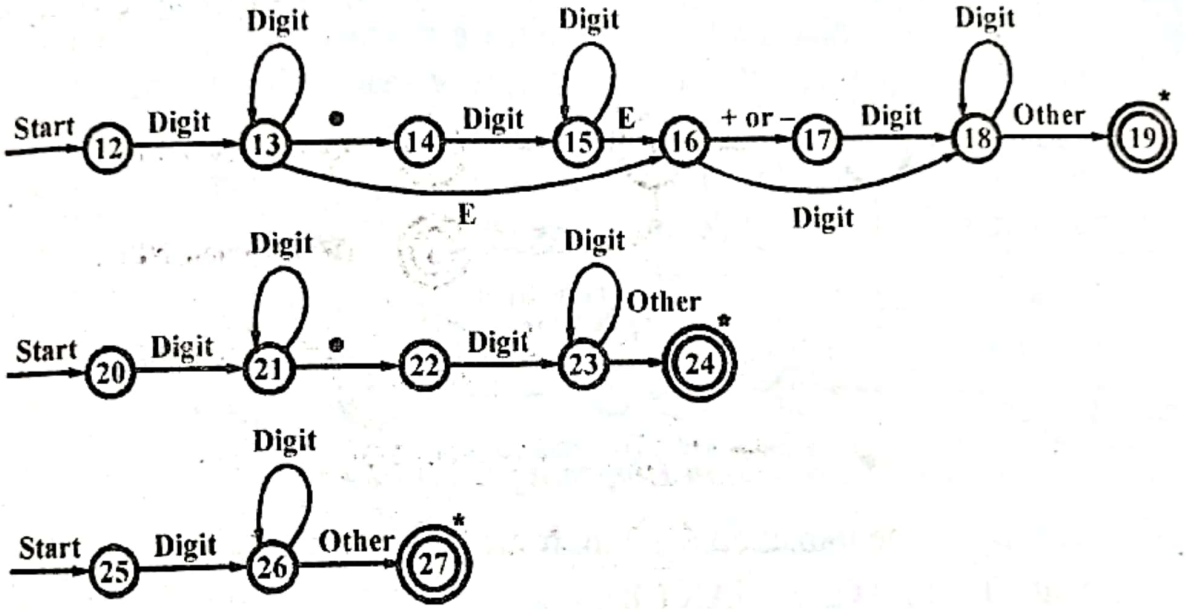


Fig. 1.25 Transition Diagrams for Unsigned Numbers in Pascal

(iii) Recognition of Number Constant – To decide for the "num" token and its nature a set of three transition diagram are followed. Firstly it is checked whether the number carries any exponential part or not. Next it is checked whether any fractional part is present or not. If not then only it is declared to be an integer constant. Thus we can separate out real number or fractional numbers or simply integer.

The action when any of the accepting states 19, 24 or 27 is reacted is to call a procedure install num that enters the lexeme into a table of numbers and returns a pointer to the created entry. Finally the token num is passed with the table reference.

(iv) Recognition of White Space – The treatment of white space as shown in fig. 1.26 is different from the other pattern because nothing is returned to the parser. We merely go back to the start state of the first transition diagram to look for another pattern.

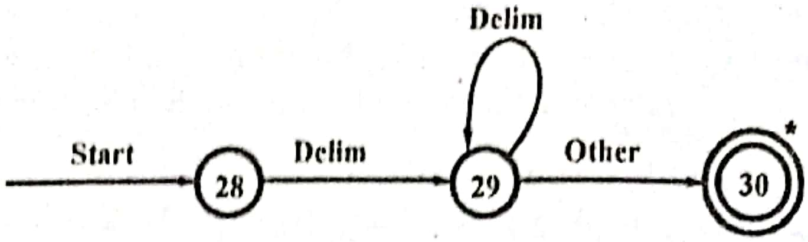


Fig. 1.26 Transition Diagram for Delimiters

Q.36. What is simple approach to the design of lexical analyzer for an identifier ? ***(R.G.P.V., Dec. 2007)***

Ans. Fig. 1.27 shows a transition diagram for an identifier, defined to be a letter followed by any number of letters or digits. The starting state of the transition diagram is state 0, the edge from which indicates that the first input character must be a letter. If this is the case, we enter state 1 and look at the next input character. If that is a letter or digit, we re-enter state 1 and look at the input character after that. We continue this way, reading letters and digits, and making transitions from state 1 to itself, until the next input character is a delimiter for an identifier, which here we assume is any character that is not a letter or a digit. On reading the delimiter, we enter state 2.

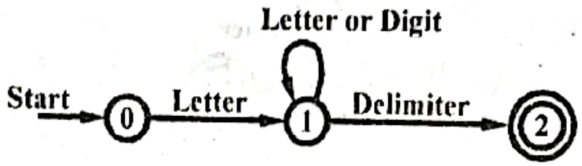


Fig. 1.27 Transition Diagram for an Identifier

Consider again the transition diagram in fig. 1.27. The code for state 0

```
State 0 : c : = GETCHAR ( );
          if LETTER (c) then goto state 1
          else FAIL ( )
```

Here, LETTER is a procedure which returns **true** if and only if c is a letter. FAIL is a routine which retracts the lookahead pointer and starts up the next transition diagram, if there is one, or calls the error routine. The code for state 1 is –

```
State 1 : c : = GETCHAR ( );
          if LETTER (c) or DIGIT (c)
              then goto state 1
          else if DELIMITER (c)
              then goto state 2
          else FAIL ( )
```

DIGIT is a procedure which returns **true** if and only if c is one of the digits 0, 1, ..., 9. DELIMITER is a procedure which returns true whenever c is a character that could follow an identifier.

State 2 indicates that an identifier has been found. Since the delimiter is not part of the identifier, we must retract the lookahead pointer one character, for which we use a procedure RETRACT. We use a * to indicate states on which input retraction must take place. In state 2 we return to the parser a pair consisting of the integer code for an identifier, which we denote by id, and a value that is a pointer to the symbol table returned by INSTALL. The code for state 2 is –

```
State 2 : RETRACT ( );
          return (id, INSTALL ( ))
```

Q.37. Explain the concept of LEX, as a lexical analyzer generator.

Or

What is LEX ? Describe auxiliary definitions and translation rules for LEX with suitable example. ***(R.G.P.V., Dec. 2005, June 2007, 2008)***

Ans. **LEX** – ***LEX*** the tool that has been widely used to specify lexical analyzers for a variety of languages. This tool is referred to as ***LEX compiler*** and input specification is called ***LEX Language***. Lexical analyzer combines the specification of pattern using regular expressions with actions e.g.–making entries into a symbol table.

A LEX source program is a specification of a lexical analyzer, consisting of a set of regular expressions together with an action for each regular expression. The output of LEX is a lexical analyzer program constructed from the LEX source specification.

Most programming languages, a source program for LEX does not supply all the details of the intended computation. Rather, LEX itself supplies with its output a program that simulates a finite automation. This program takes a transition table as data. The transition table is that portion of LEX's output that stems directly from LEX's input. The situation is depicted in fig. 1.28.

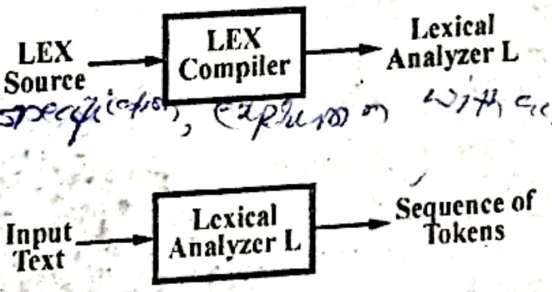


Fig. 1.28 The Role of LEX

Auxiliary Definitions – A LEX source program consists of two parts, a sequence of ***auxiliary definitions*** followed by a sequence of ***translation rules.*** The auxiliary definitions are statements of the form

$$D_1 = R_1$$
$$D_2 = R_2$$
$$\vdots$$
$$D_n = R_n$$

Where each D_i is a distinct name, and each R_i is a regular expression whose symbols are chosen from $\Sigma \cup \{D_1, D_2, \ldots\ldots, D_{i-1}\}$, i.e., characters of previously defined names. The D_i's are shorthand names for regular expressions. Σ is our input symbol alphabet, e.g., the ASCII or EBCDIC character sets.

We can define the class of identifiers for a typical programming language with the following sequence of auxiliary definitions.

$$\text{letter} = A \mid B \mid \ldots\ldots \mid Z$$

$$\text{digit} = 0 \mid 1 \mid \ldots\ldots \mid 9$$

$$\text{identifier} = \text{letter (letter} \mid \text{digit)}^*$$

Translation Rules – The translation rules of a LEX program are statements of the form.

P_1 {action 1}
P_2 {action 2}
.....
P_n {action n}

where each P_i is a regular expression called pattern and each action ***action i*** is a program fragment that is to be executed whenever a Lexeme matched by pattern P_i is found in the input. The ***action i***'s are written in a conventional programming language; To create the lexical analyzer L, each of the ***Action i***'s must be compiled into machine code, just like any other program written in the language of the ***action i.***

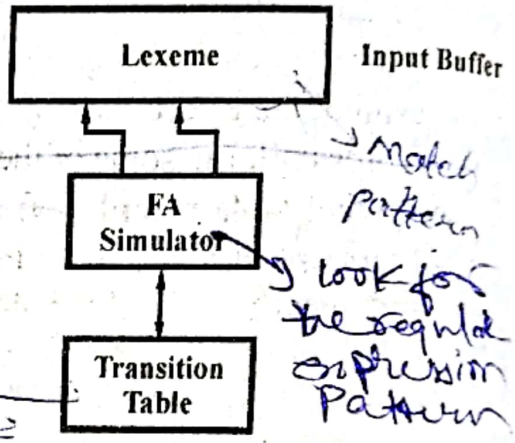


Fig. 1.29 Model of LEX Compiler

The LEX source program also consists a third section known a auxiliary procedures. This section holds whatever auxiliary procedures are needed by the actions. These procedures can be compiled separately and loaded with the lexical analyzer.

One problem is to construct a recognizer that looks for lexemes in the input buffer. If more than one pattern matches, the recognizer is to choose the longest lexeme matched. If there are two or more patterns that match the longest lexeme, the first listed matching pattern is chosen.

There is an input buffer, in the model of the LEX compiler, with two pointers to it these are – a lexeme beginning and a forward pointer. The lexical compiler constructs a transition table for a finite automation from the regular expression pattern in the LEX specification. The lexical analyzer itself consists of a finite automation simulator that uses this transition table to look for the regular expression patterns in the input buffer.

Q.40. Develop a lexical analyzer to recognize valid operators of C program.
(R.G.P.V., Dec. 2006)

Ans. In the declarations section, declarations are surrounded by the special brackets % {and %}. Anything appearing between these brackets is copied directly into the lexical analyzer, and is not treated as part of the regular definitions or the translation rules. There are two procedures, ***install_id*** and ***install-num***, that are used by the translation rules.

A lexical analyzer that recognize valid operators of C program is as follows –

```
%{
    /* definitions of mainfest constants
       LT, LE, EQ, NE, GT, GE,
       IF, THEN, ELSE, ID, NUMBER, RELOP */
% }
/* regular definitions */
delim          [ \t \n ]
ws             {delim} +
letter         [A – Z a – z]
digit          [0 – 9]
id             {letter} ({letter} : {digit}) *
```

```
number          {digit} + (\.{digit}+) ? (E[+\-] ? {digit} +) ?
%%
{ws}            {/* no action and no return */}
if              {return (IF);}
then            {return (THEN);}
else            {return (ELSE);}
{id}            {yylval = install_id ( ); return (ID;}
{number}        {yylval = install_num ( ); return (NUMBER);}
" < "           {yylval = LT; return (RELOP);}
" < = "         {yylval = LE; return (RELOP);}
" = "           {yylval = EQ; return (RELOP);}
" < > "         {yylval = NE; return (RELOP);}
" > "           {yylval = GT; return (RELOP);}
" > = "         {yylval = GE; return (RELOP);}
%%
Install_id ( ) {
    /* procedure to install the lexma, whose first character is pointed to
by yytext and whose length is yyleng, into the symbol table and return a
pointer there to */
}
install_num ( ) {
    /* Similar procedure to install a lexeme that is a number */
}
```

NUMERICAL PROBLEMS

Prob.1. A Lex program is given below –

Auxiliary Definitions

(non)

Translation rules

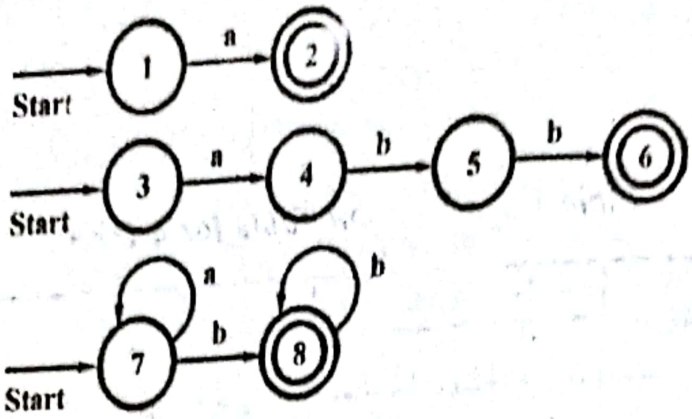
a

abb

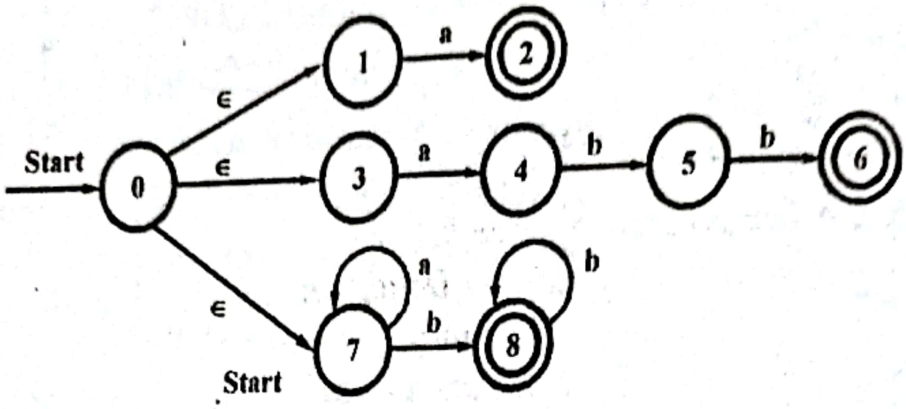
a^*b^+

Implement the LEX program as DFA. ***(R.G.P.V., Dec. 2003)***

Sol. The three tokens are recognized by the automata in fig. 1.31 (a).



(a) NFA for a, abb, and a^*b^+



(b)

Fig. 1.31 NFA Recognizing Three Different Patterns

The string aaba as input string is accepted by N as follows –

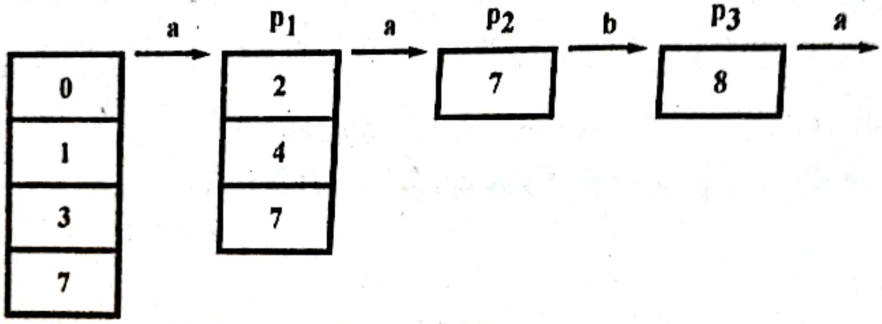


Fig. 1.32 Sequence of Sets of States Entered in Processing Input aaba

Fig. 1.32 shows the sets of states and patterns that match as each character of the input aaba is processed. The initial set of states is {0, 1, 3, 7}. States 1, 3 and 7 each have a transition on a, to states 2, 4 and 7, respectively. Since state 2 is the accepting state for the first pattern, we record the fact that the first pattern matches after reading the first a. For the second input the transition from state 7 to state 7. There is a transition from state 7 to 8 on the input character b. Once we reach state 8, there are no transitions possible on the next input character a so we have reached termination. Since the last match occured after we read the third input character, we report that the third pattern has matched the lexeme aab.

When we convert an NFA to a DFA using the subset construction Algorithm, there may be several accepting states in a given subset of nondeterministic states. In such a situation, the accepting state corrosponding to the pattern listed first in the Lex specification has priority.

Table 1.5 Transition Table for a DFA

State	Input Symbol		Pattern Announced
	a	b	
0137	247	8	none
247	7	58	a
8	–	8	a* b*
7	7	8	none
58	–	68	a* b*
68	–	8	abb

Prob.2. A Lex program is given below –

Auxiliary Definition

(non)

Translation rules

a

abb

a*b*

Implement the Lex program as DFA. ***(R.G.P.V., June 2004)***

Sol. Refer the similar approach in Prob. 1.

Prob.3. Write a LEX program for the following strings –

w.s (white space), if, then, else, id

(R.G.P.V., June 2003)

Or

Write a LEX programme to recognize white space, identifier, if, then and else. ***(R.G.P.V., Dec. 2008)***

Sol. A Lex program that recognizes the tokens and returns the token found. A few observations about the code will introduce us to many of the important features of Lex.

A Lex program is given below –

```
% {
    /* definitions of manifest constants
    IF, THEN, ELSE, ID */
% }
    /* regular definition */
```

```
        delim       [\t\n]
        ws{delim} +
        letter      [A-Z, a-z]
        id          {letter} ({letter} ! {digit})
    % %
        {ws}        {/* no action and no return */}
        if          {return (IF);}
        then        {return (THEN);}
        else        {return (ELSE);}
        {id}        {yylval = install_id( ); return (ID);}
    % %
        install_id {
        /* procedure to install the lexeme, whose first character is
        pointed to by yytext and whose length is yyleng, into the symbol
        table and return a pointer there to */
        }
```