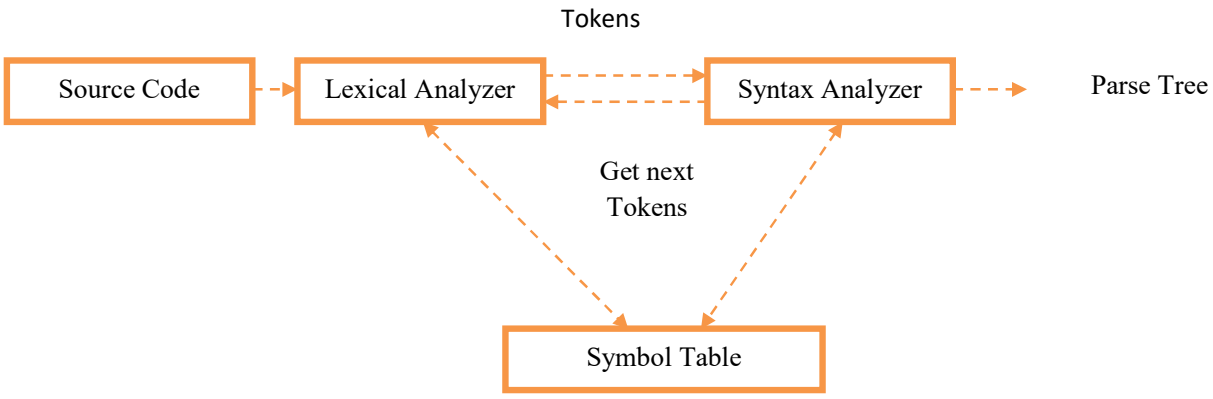
UNIT-III

Lexical Analysis

Role of Lexical Analyzer:

- The main role of lexical analyzer is to convert source code into number of tokens.
- Lexical Analysis is the first phase of compiler.
- It takes the source code that is written in the form of sentence.
- The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespaces or comments in the source code.
- If the lexical Analyzer finds a token invalid it generates an error.
- The lexical analyzer works closely with syntax analyzer.
- It reads character streams from the source code, checks the legal tokens and passes the data to the syntax analyzer when it demands.



Working of Lexical Analyzer

Tokens:

- Lexems are said to be sequence of characters in a token.
- There are some predefined rules for every lexems to be identified as valid tokens.
- These rules are defined by grammar rules, by means of pattern.
- A pattern explains what can be a token and these patterns are defined by means of regular expressions.
  e.g.
  int value=100;
  contains the tokens
  int (keyword), value (identifire), = (operator), 100(constant) and ;(symbol)

- The lexical analyzer also follows le priority where a reserved word e.g. keyword of a language is given priority over user input i.e. If lexical analyzer finds a lexeme that matches any existing reserved word then it should generate an error.

Specification of Tokens:

a) Alphabets:
Any finite set of symbols {0, 1} is a set of binary alphabets.
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F} is a set of hexadecimal alphabets.
{a-z, A-Z} is a set of English language alphabets.

b) Strings:
Any finite sequence of alphabets is called a string.
Length of the string is the total number of occurrences of alphabets.

c) Special Symbol:
Arithmetic symbol ⇨ +, -, *, /, %
Punctuation ⇨ (,) comma, semicolon (;), dot (.), arrow (→)
Assignment ⇨ =
Special Assignment ⇨ +=, -=, *=, /=

Comparison ⇨ >, <, >=, <=, !=, ==

Pre-processor⇨ #
Location specifies⇨ &
Logical⇨ &, &&, ||, !
Shift operator ⇨ >>, >>>,<<, <<<

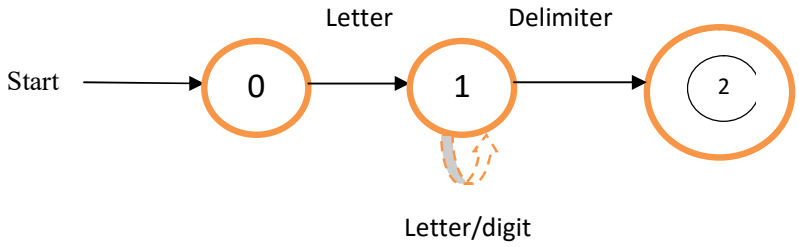
d) Language:

A language is considered as a finite set of strings over some finite set of alphabets.

Simple Approach to the Design of Lexical Analysis:

- One way to design any program is to describe the behaviour of program by using flowchart.
- For implementing lexical analyzer we need to describe the behaviour of the program by using specialized flowchart for lexical analyzer is called as transition diagram.
- In transition diagram, the boxes of flowchart are drawn by using circles called as states.
- The states are connected by an arrows called as Edges.
- The double circles in any transition diagram denote its final states.

Eg.



Transition diagram for identifier.

- It is a collection of letters and digits.
- The starting state of the diagram is the initial state denotes by 0.
- The edge indicates that the first character is always forming the letter only.

- If the next character is letter or digit then next state i.e. state 1 is repeated.
- Continue this way until the next input character is delimiter.
- To convert transition diagram into a program i.e. we want to construct segment of code for each state
- For this purpose we use GETCHAR() function.
- The next step is to found then control is transfer to next edge/step.
- If such edge is not found then state is considered as end point.

Consider the transition diagram. For state 0 and state 1 is as follows:

State 0:
C:= GETCHAR();
If letter (c) then go to state 1;
else
FAIL();

State 1:
C:= GETCHAR();
if letter(c) then go to state 1;
else if delimiter (c) then go to state 2;
else fail();

Regular expression:

- Lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belongs to the language in hand.
- It searches for the pattern defined by the language rules.
- Regular expressions have the capability to express finite string of symbols.
- The grammar defined by regular expressions is called as regular grammar.

- The language defied by regular grammar is known as regular language.
- Regular expression is an important notation for specifying grammar patterns.
- Each pattern matches set of strings, so regular expressions serve as names for a set of strings.
- Programming language tokens can be described by regular languages.
- Regular languages are easy to understand and have efficient implementation.
- There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations:

The various operations are performed on regular languages are:

1) Kleene closure:

The kleene closure of a language L is written as

L^* = zero or more occurrences of language L

A^*={€, 1, 2, 3,.....}

L^*={€, L, LL, LLL, LLLL, ...}

2) Positive closure:

The positive closure of a language L is written as

L^+={ L, LL, LLL, LLLL, ...}

3) Concatenation:

Concatenation of two languages i.e. L and M is written as

LM={st | s is in L and t is in M}

4) Union:

Union of two languages i.e. L and M is written as

L∪M={s | s is in L or t is in M}

L∪M={L, M}

Notations:

It r and s re regular expressions denoting the language L(r) and L(s) then

Union:

(r)|(s) is a regular expression denoting L(r) ∪ L(s)

Concatenation:

(r)(s) is a regular expression denoting L(r) L(s)

Kleene closure:

(r)* is a regular expression denoting (L(r))*

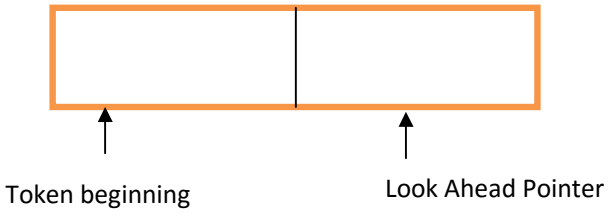
(r) is a regular expression denoting L(r)

Tokens can be described by using regular expression is as follows:

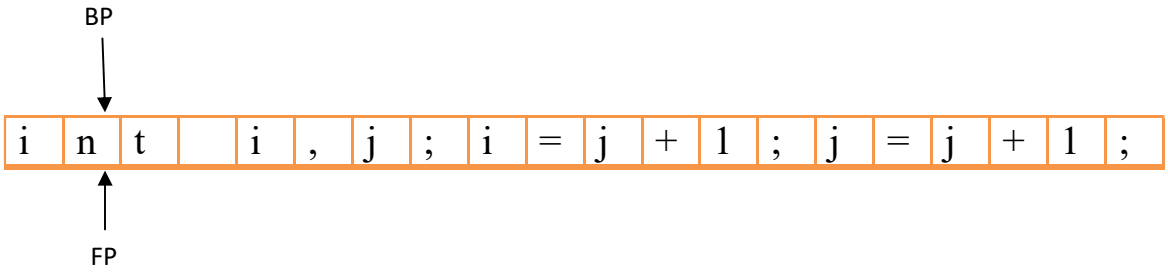
a) Keyword – if |else | float |int
b) Identifier – if |else| float |int
c) Relational operator - ≥ | ≤ | == | !=
d) Constant – (digits)*

Input buffering:

- The input buffering is a location that holds on incoming information.
- The lexical analyzer scans the characters of source program one at a time.
- The many character has to ne examine before the next tokens can be determine therefore the lexical analyzer read its input from input buffer.



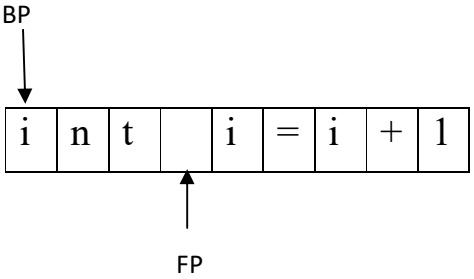
- The token beginnings specify the beginning of the tokens.
- The lookahead pointer scans the source code until token is discovered.
- The lexical analyzer scans the input from left to right one character at a time.
- It uses two pointer beginning ptr (BP) and forward to keep track of the pointer of the input scanned.



Initially both the pointer point to the first character of the input string.

One buffer scheme:

In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme, that makes overwriting the first of lexeme.



One buffer scheme storing input string

Finite Automata:

- Finite Automata is a state machine that takes a string of symbols as input and changes its state accordingly.
- Finite Automata is a recognized for regular expression.
- The finite Automata is a program which takes input as string x and answer yes if x is input of L and answer no if x is not input of L
- The finite Automata is a recognizer which simply say yes or no

The mathematical model of finite automata consists of:

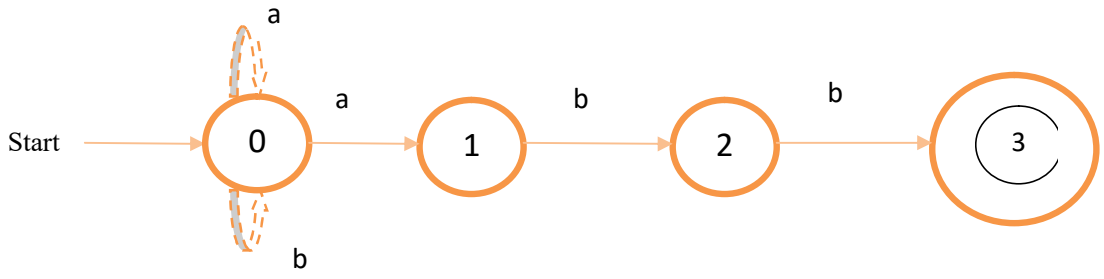
a) **Finite set of states(Q)**
b) **Finite set of input symbols(∑)**
c) **Start state(q_0)**
d) **Set of final states(q_f)**
e) **Transition function (δ)**

The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (∑) i..e. $Q X \sum \rightarrow Q$

The finite automata having two types:

a) Non- deterministic finite Automata (NFA):

- for implement regular expression we construct specialized transition diagram called as non- deterministic finite automata(NFA)
- The NFA recognizing the language (a/b)* is as follows:



- The NFA is look like transition diagram.
- The NFA nodes are called as states and edge are called as transitions.
- NFA can be represented in table format. The table is called as transition table.
- Table for the above transition diagram is :

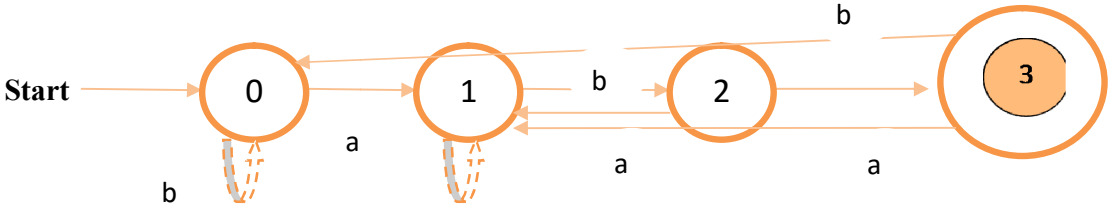
State	Input Symbol	
	a	b
0	{1, 0}	{0}
1	-	{2}
2	-	{3}

Transition Table

The above NFA will accept input as abb, aabb, babb, aaabb.
e.g The string aabb is accepted from state 0
The next input accepted from state 0, 1, 2, 3

b) Deterministic Finite Automata(DFA):

- The NFA has more than one transition from state 0 for same input A that is it may go from state 0 and state 1.
- It means the state 0 having two transitions for same input therefore it is difficult to handle NFA so that deterministic version of finite Automata developed.
- We describe finite automata is deterministic
  If
  i) it has no transition on same input
  ii) for each step it has almost one transition
- The deterministic finite automata for (a/b)* is as follows



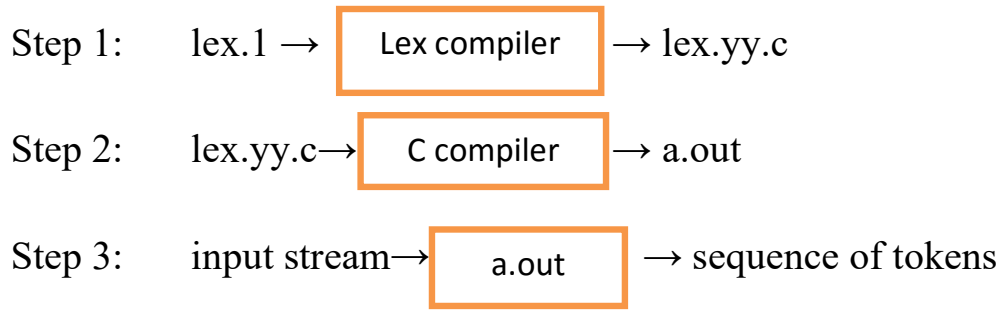
Transition table for above transition diagram is:

State	Input Symbol	
	a	b
0	{1}	{0}
1	{1}	{2}
2	{1}	{3}
3	{1}	{0}

The above table DFA will accept the input string as abb, aabb, aabbb, ababb.

#A Language for specifying Lexical Analyzer:

- A tool called lex or flex allows us to specify lexical analyzer.
- A lex tool or compiler which, automatically generate lexical analyzer.
- The input for lex is called as lex language.
- The lex compiler converts input pattern into transition diagram.
- The lex language is denoted by lex.1 and store it into the file called as lex.yy.c
- Working of lex tool



- An input file called lex.1 is written in lex language.
- The lex compiler convert lex.1 into a file known as lex.yy.c
- A file is again compiling using C compiler into output file called a.out.
- a.out takes input stream and produce sequence of tokens.

The lex source program consisting of two parts:

a) Auxiliary functions:

It holds additional functions used in lex.

b) Translation rules:

DN=RN

DN is name for regular expression and RN is rule for regular expression.

e.g.:

identifire=letter(letter/digit)*

letter(a-z | A-Z)

digit(0-9)

Structure of lex program is

Declaration ⟹ {identifire/digits/variable}

%% include declaration of variable

Translation rules ⟹ Rules for regular expression

%%

Auxiliary function