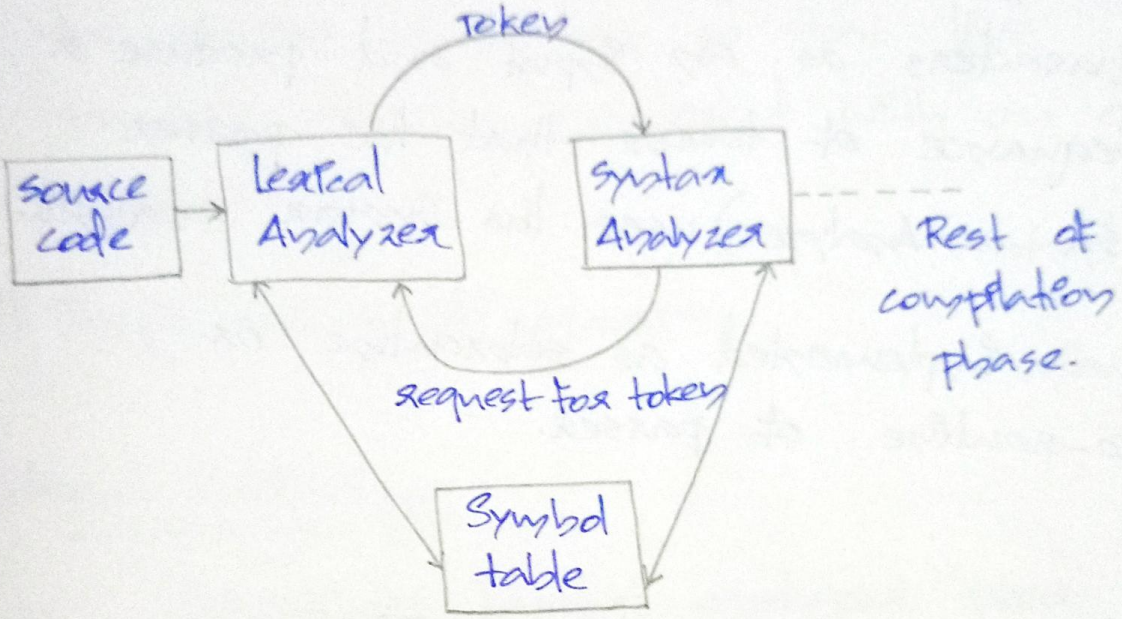
Module :2

Lexical Analysis :-

Lexical analyzer is the first phase of the compilation also known as scanner. It converts the high level input program into a sequance of tokens



functions of lexical Analyzer :-

1. Read input characters from source program.
2. Dividing the program into valid tokens.
3. Remove white spaces characters.
4. Remove comments (//, *).
5. Helps to identify token into the symbol table.

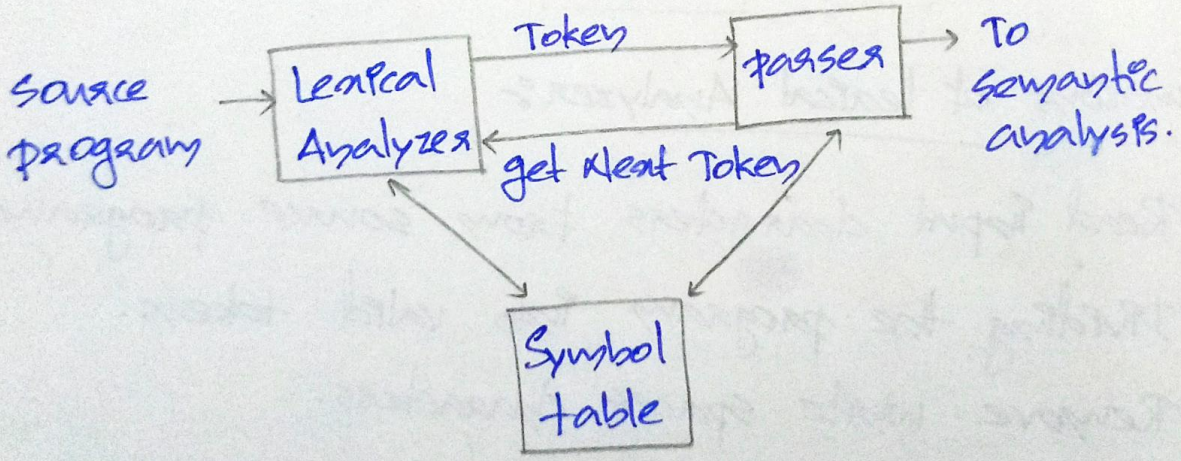
⑥ It generate lexical errors.

* Role of lexical Analyzer:

① Lexical analyzer is the first phase of a compiler.

② Main task is to read a stream of characters as an input and produce a sequence of tokens that the parser (Syntax Analyzer) uses for Syntax analysis.

③ Usually implemented as subroutine or co-routine of parser.



④ Seperation of the input source code into tokens.

⑤ stripping out the comments and unnecessary white spaces in the form of blank, tab and newline from the source code.

⑥ keeping track of line numbers while scanning the newline characters so that line numbers can be associated with an error message.

⑦ Preprocessing of Macros.

Input Buffering:-

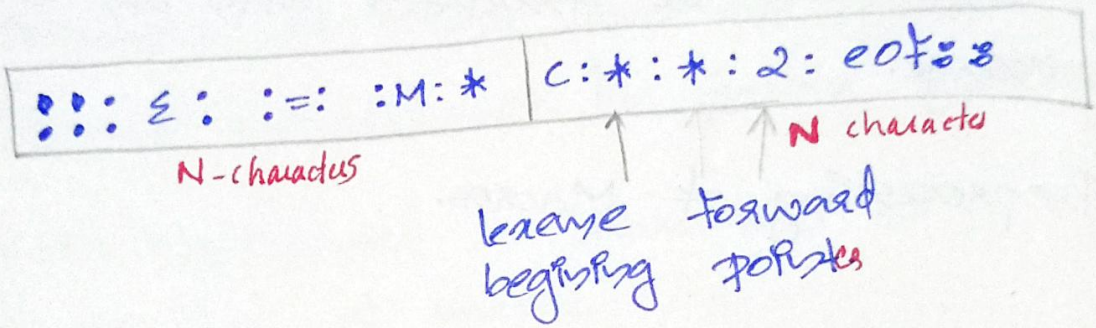
Sometimes Lexical analyzer need to look a head several characters beyond the lexeme for a pattern before a match can be announced.

* As large amount of time can be consumed moving characters we need two buffers scheme to handle large look a head

safety.

* Buffer is divided into two N-characters halves. Where N is the number of characters on one disk block.

* Read N input characters into each half of the buffer with one system read command.



* If fewer than N characters remain in the input, a special character, eof marks the end of source file.

* Two pointers to the input buffer are maintained.

* String of characters between the two pointers is the current lexeme.

* Initially both pointers point to the first character of the next lexeme to be found.

* One called the forward pointer scans ahead until a match for a pattern is found.

* Once the next lexeme is determined, the forward pointer is set to the characters at its right end.

* If the forward pointer is about to move past the halfway mark, right half is filled with y new input characters.

* If the forward pointer is about to move past the right end of the buffer, the left half is filled with y new characters and the forward pointer wraps around to the beginning of the buffers.

* After the lexeme is processed both pointers are set to the characters

Immediately past the lexeme.

* code to advance forward pointer:

  If forward at end of first half then begin reload second half:

  ```
  forward = forward + 1
  ```

  end

* else if forward at the end of second half then begin reload first half:

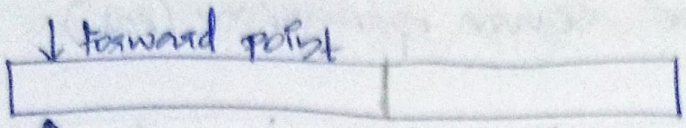
  ```
  Move forward to beginning of first half
  ```

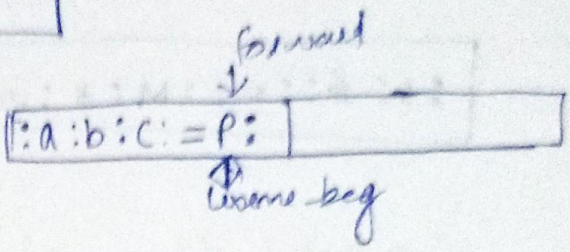
  end

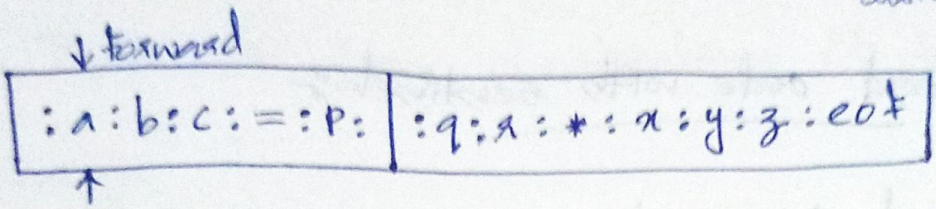
* else

  ```
  forward = forward + 1
  ```

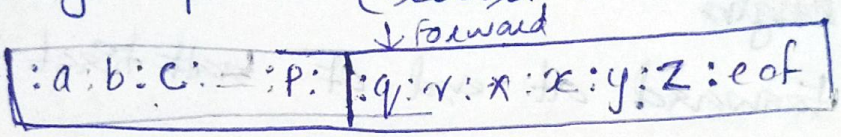
Eg: abc - pqr * xyz







abc = Identifier. (first lexeme)
= assign operator (second lexeme)



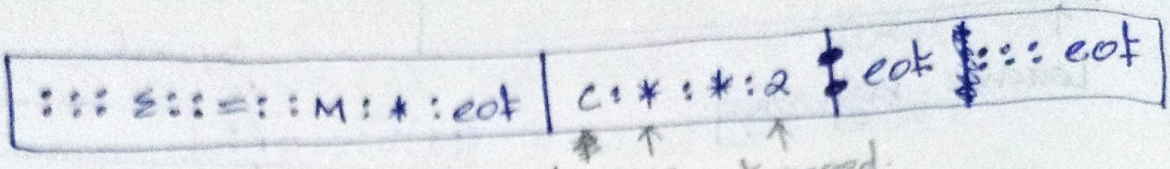
Sentinels:-

Except at the end of the buffer halves, the above code requires two tests for each advance of the forward pointer.

* In first model, we want to test after every incrementation of FP
* We can reduce the two tests to one. If we extend each buffer half to hold a

sentinel characters at the end.

* Sentinel is a special character that cannot be part of the source program (eof)



* Look ahead code with sentinels:

```
forward = forward + 1;
① If  forward = eof
  then begin
       ~~begin~~
  If forward at end of ~~both~~ first half
  then
       begin reload second half
       forward = ~~forward~~ forward + 1
  end
  else If forward at end of second half
  then
       begin
```

related text book:

move forward to beginning of text book

move forward to beginning etc

Specifications of Tokens:-

There are three specifications of tokens:-

1. Strings.
2. Languages.
3. Regular expressions.

1. Strings:-

An alphabet or character class is a finite set of symbols. A string over an alphabet is a finite sequence of symbols drawn from that alphabet.

a, b → alphabet

aa
ab
ba
bb
} → Strings

2. Language:-

A Language is any countable set of strings over some ~~finite alphabet~~ fixed alphabet.

* In language theory, the terms sentence and word are often used as synonyms for string. The length of a string s, usually written |s|, is the number of occurrences of symbols in s.

Eg:- banana is a string of length 6.

* The empty string denoted as ε, is the string of length zero.

* Symbols = {0, 1}

String = 0, 1, 01, 10, 11, 00, 111, 100, 101

Language = {1, 11, 111, 1111}

* Operations on strings:-

The following string related terms are commonly used:-

① A prefix of string s is any string obtained

by removing zero or more symbols from the end of string s.

Eg: ban is a prefix of banana.

② A suffix of string s is any string obtained by removing zero or more symbols from the beginning of s.

Eg: nana is a suffix of banana.

③ A substring of s is obtained by deleting any prefix and any suffix from s.

Eg: ban is a ~~star~~ substring of banana.

④ The proper prefix, suffixes and substrings of a string s are those prefixes, suffixes and substrings of s that are not ε (empty) or not equal to s itself.

⑤ A subsequence of s is any string formed by deleting zero or more not necessarily

consecutive positions of s

Eg: banan is a subsequence of banana.

* Operations on language:

The following are the operations that can be applied to languages.

① Union.
② Concatenation.
③ Kleene closure.
④ Positive closure

* The following examples shows the operations on strings:

Let, L = {0, 1} and S = {a, b, c}

① Union:

L ∪ S = {0, 1, a, b, c}

② Concatination:

L.S = {0a, 1a, 0b, 1b, 0c, 1c}

③ kleene closure: (0 or More Occurrence)

$L^* = \{\epsilon, 0, 1, 00, 11, 10, 01 \ldots\}$

④ Positive closure:

$L^+ = \{0, 1, 00, 11, 10 \ldots\}$

It is similar to kleene closure except ϵ.

Regular expressions:-

Each regular expression r denotes a language $L(r)$. Here the rules that define the regular expressions over some alphabet Σ and the languages that expression denotes:-

① ϵ (epsilon) is a regular expression and $L(\epsilon)$ is $\{\epsilon\}$ (empty), that is the language whose set member is the empty string.

② If 'a' is a symbol in Σ (sigma) then 'a' is a regular expression and

$L(a) = \{a\}$. That is the language with one string of length one with 'a' in its one position.

③ Suppose r and s are regular expressions denoting the language $L(r)$ and $L(s)$. Then,

(i) $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.

(ii) $(r).(s)$ is a regular expression denoting the language $L(r).L(s)$.

(iii) $(r)^*$ is a regular expression denoting $(L(r))^*$.

(iv) (r) is a regular expression denoting $L(r)$.

④ May operator * has highest precedence and is left associative concatination has second precedence.

④ Regular definitions

Giving names to regular expressions is referred to as a regular definition.

* If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form.

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

$d_n \rightarrow r_n$

* d_1, d_2, d_n are the names given to the regular expression (definition).

① Each d_i is a distinct name.

② Each r_i is a regular expression over the alphabet $\Sigma \cup \{d_1, d_2, \ldots d_{i-1}\}$

In this case
r_2 can use d_1
r_3 " d_2
r_4 " d_3

~~Examples~~

Eg: identifiers (used in pascal)

regular definition:

letters → A|B|--|z| |a|b|...|z|

digit → 0|1|2 . . |9

id → letter(letter|digit)*

$d_1 \to$ letter (r_1)
$d_2 \to$ digit (r_2)
$d_3 \to$ id (r_3)

$id \to r_1(r_1|r_2)^*$

Eg: unsigned number (5280) 39.37, 6.336E4

digit → 0|1|2 . . |9

digits → digit digit*

$d_1 \to r_1$
$d_2 \to r_2$

optional fraction → .digits | ∈

$d_3 \to r_3$

optional exponent → (E(+|-|∈)digit) | ∈

$d_4 \to r_4$

number → digits optional fraction optional exponent.

$d_5 \to r_5$

Regular sets

A language that can be defined by a regular expression is called a regular set. If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s.

* There are a number of algebraic laws on regular expressions that can be used to manipulate them into equivalent forms.

* For instance, r|s = s|r is commutative.

r((s|t)) = ((r|s)|t)

r((s|t))
(r(s)|t)

Short hands:

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational short hand for them.

① One or more instances (+):-

The unary postfix operator + means * one or more instances.

* If r is a regular expressions that denotes the language L(r), then (r)+ is a regular expression that denotes the language (L(r))+.

* Thus the regular expression a^+ denotes the set of all strings of one or more a's

* The operator + has the same precedence and associativity as the operator *.

② Zero or one instance (?):-

The unary postfix operator ? means zero or one instance.

* The notation $r?$ is a shorthand for $r|\epsilon$.

* If r is a regular expression, then $L((r)?)$ is a regular expression that denotes the language.

Character classes:-

The notation [abc] where a, b, and c are alphabet symbols ~~the~~ denotes the regular expression a|b|c.

* The character class such [a-z] denotes the regular expression a|b|c|d|...|z.
* We can describe identifiers as being strings generated by the regular expression,

[A-z a-z] [A-Za-z0-9]*

Eg: ① Identifiers

letter → [A-Za-z_]

digit → [0-9]

id → letter-(letter-|digit)*

② unsigned number.

digit → [0-9]

digit → digit+

number → digits (. digits)? (E [+ -]? digits)?

Recognition of tokens using finite automata:

Indicate an accepting state by a double circles, and if there is an action to be taken typically returning a token and an

attribute value to the parser.

* If it is necessary to retract the forward pointer one position (that is, the lexeme does not include the symbol that got us to the accepting state). The we shall additionally place a * near that accepting state.

* One state is designated the start state, or initial state, it is indicated by an edge, labled 'start' entering from now where.

* The transition diagram always begins in the start state before any input symbols have been read.

* Tokens can be recognized with the help of transition diagrams.

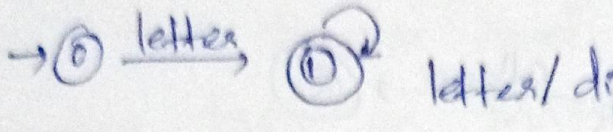
Eg:- Recognition of tokens (identifiers):-

letter → a|b| -- |z| A|B| -- |Z|

digit → 0|1|2| -- |9

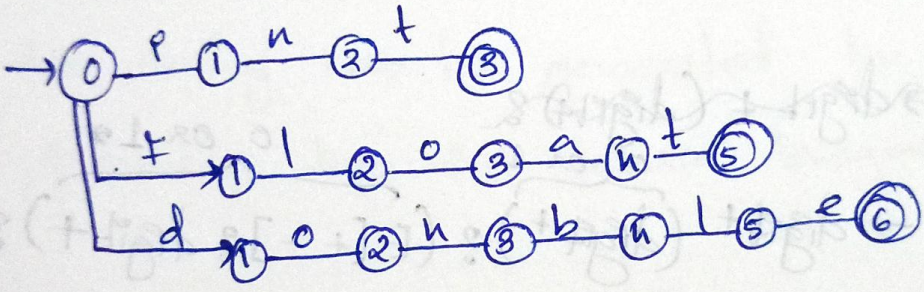
id → letter (letter/digit)*

∴ Transition diagram,



② Recognition of keywords:

We can use transition diagrams to recognizing keywords like int, float, double, for, if etc.



③ Recognition of variables:-

We can use - at beginning, a digit at the later part. So,

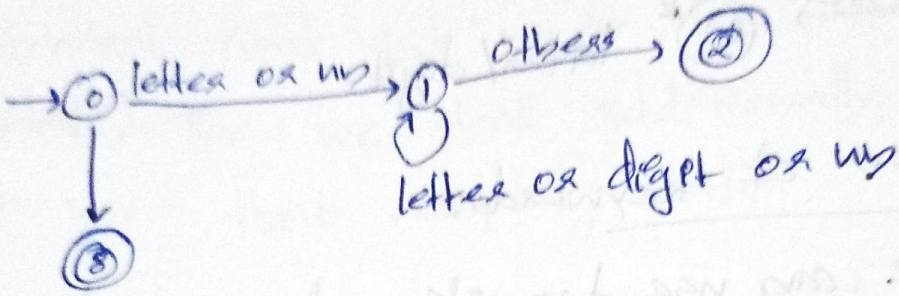
un → —

letter → a|b|c|--|z| A|B|---|Z|

digit → 0|1|2|--|9|

var → (letter|_)* (letter|digit|_)*

for var,



④ Recognition of constants:-

Eg:- 123, 123.45

~~number → digit + (digit+)?~~

number → digit+ (digit+)? (E[+-]? digit+)?

0 or 1 — (digit+)?

0 or 1 — (E[+-]? digit+)?

digit [(.digit+)]? (E[+-]? digit+)?

0,1 — [(.digit+)]?

0,1 — (E[+-]? digit+)?