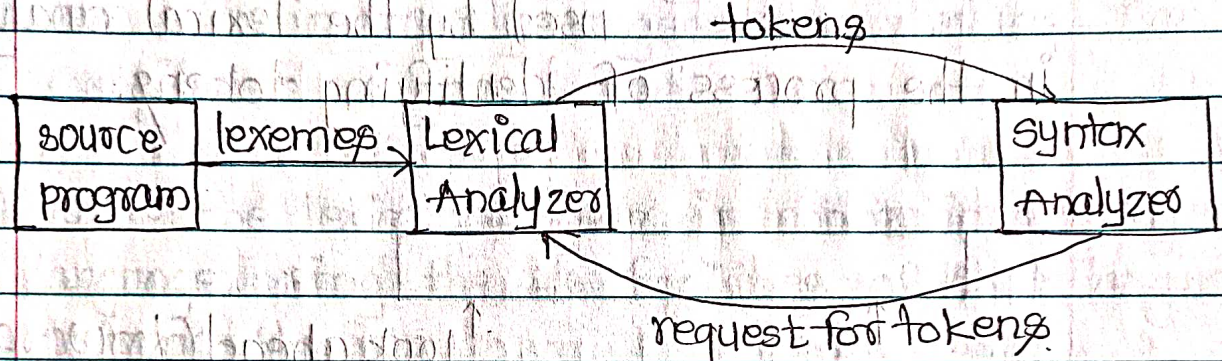
Chapter - 07

Lexical Analysis

The scanner for practical implementation is a co-routine that often works with the parser & is called whenever the parser needs a new token.



In programming language, keywords, constants, identifiers, strings, numbers, operators & punctuations symbols can be considered as tokens.

ex. int value = 100;

<keyword, int> <operator, = >
<identifier, value> <constant, 100>
<symbol, ;>

keyword	identifier	operator	constant	symbol

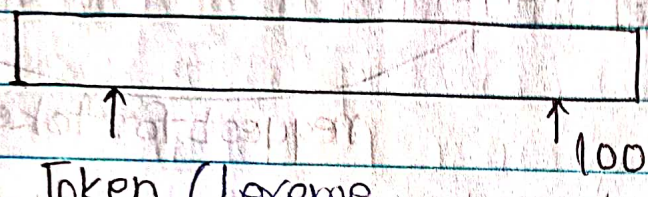
As soon as scanner sends token to syntax analysis pha the scannor get suspended. Syntax checker do the grammer check over the token & then asks for next token.

Above process is recursive till the entire input string is consumed.

Token	Patterns	Lexemes
id	letter followed by letters & digits	pi, count, D2

lexeme - meaningful sequence of characters.

Input Buffering :- It is the memory area used by the lexical analyzer in the process of identifying tokens.



Two pointer used here :-

1. One marking the begining of the token being discovered
2. lookahead pointer, which scans ahead of the beginning pointer until the token is discovered.

* The Techniques used to implement lexical analyzers can also be applied to other areas such as query languages & information retrieval system
* In lex programming language, patterns are specified by regular expression, and a compiler for lex can generate an efficient finite-automaton recognizer for the regular expressions.

There are 3 general approaches to the implementation of a lexical analyzer:-

1] Use a lexical analyzer generator, such as - Lex Compiler.

2] Write the lexical analyzer in conventional programming language, using the I/O facilities of that language to read the input.

3] Write the lexical analyzer in assembly language & explicitly manage the reading of input.

Specification of Tokens:-

How to specify the token?

- By using regular expression.

Recognition of Tokens:-

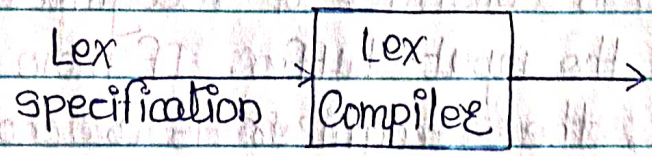
Task of Lexical Analyzer.

1) Stripping out comments & whitespace (blank, newline, tab etc).
2) Correlating error messages generated by the compiler with the source program. For ex: The lexical analyzer may keep track of the number of newline character seen, so it can associate a line number with each error message.
3) If the source program uses a macro-preprocessor the expansion of macros may also be performed by the lexical analyzer.

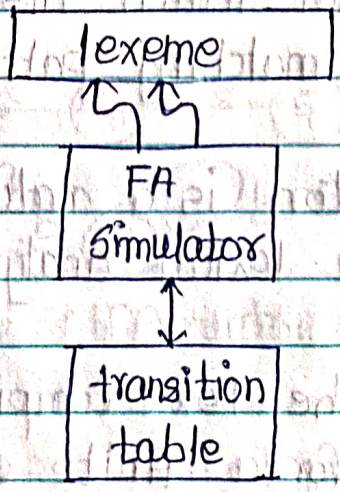
Design of a Lexical Analyzer Generator:-

OR

Design of Lex Compiler



ⓐ Lex Compiler.



ⓑ Schematic lexical analyzer.

- We assume that we have a specification of a lexical analyzer of the form

P1	{action1}
P2	{action2}
---	---
Pn	{actionn}

- Where, Pi is a regular expression & each action actioni

is a program fragment that is to be executed whenever a lexeme matched by Pi is found in the input.

- Our 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.

- A finite automaton is a natural model around which to build a lexical analyzer.

- From above fig. The lex compiler construct a transition table for a finite automaton from the regular expression patterns in the lex specification.

- The lexical analyzer itself consist of a finite automaton simulator that uses this transition table to look for the regular expression patterns in the input buffer.

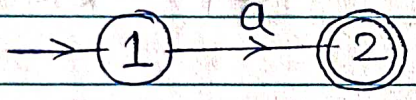
- The implementation of a Lex compiler can be based on either nondeterministic or deterministic automaton

Patteern Matching Based on NFA's

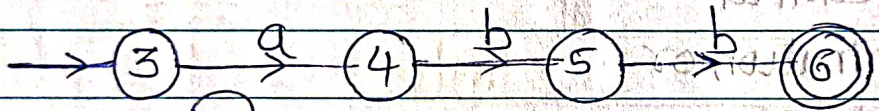
ex. Consider following lex specification.

```
a       {   }   /* Empty action */
abb     {   }
a*b+    {   }
```

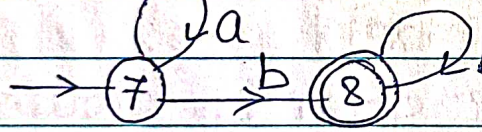
- Construct NFA for each regular expression.



RE = a.

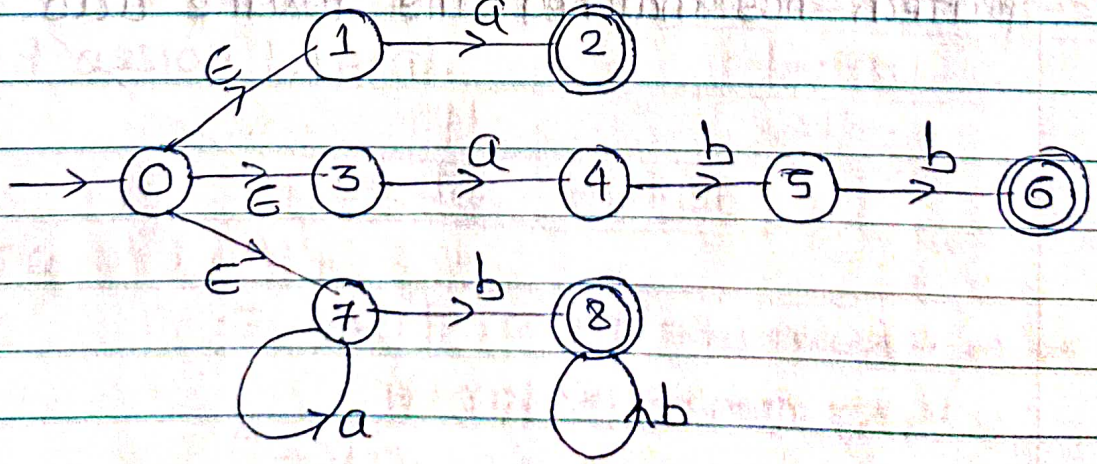


RE = abb



RE = a*b+

Combine all NFA into single NFA as follow.



Combined NFA.