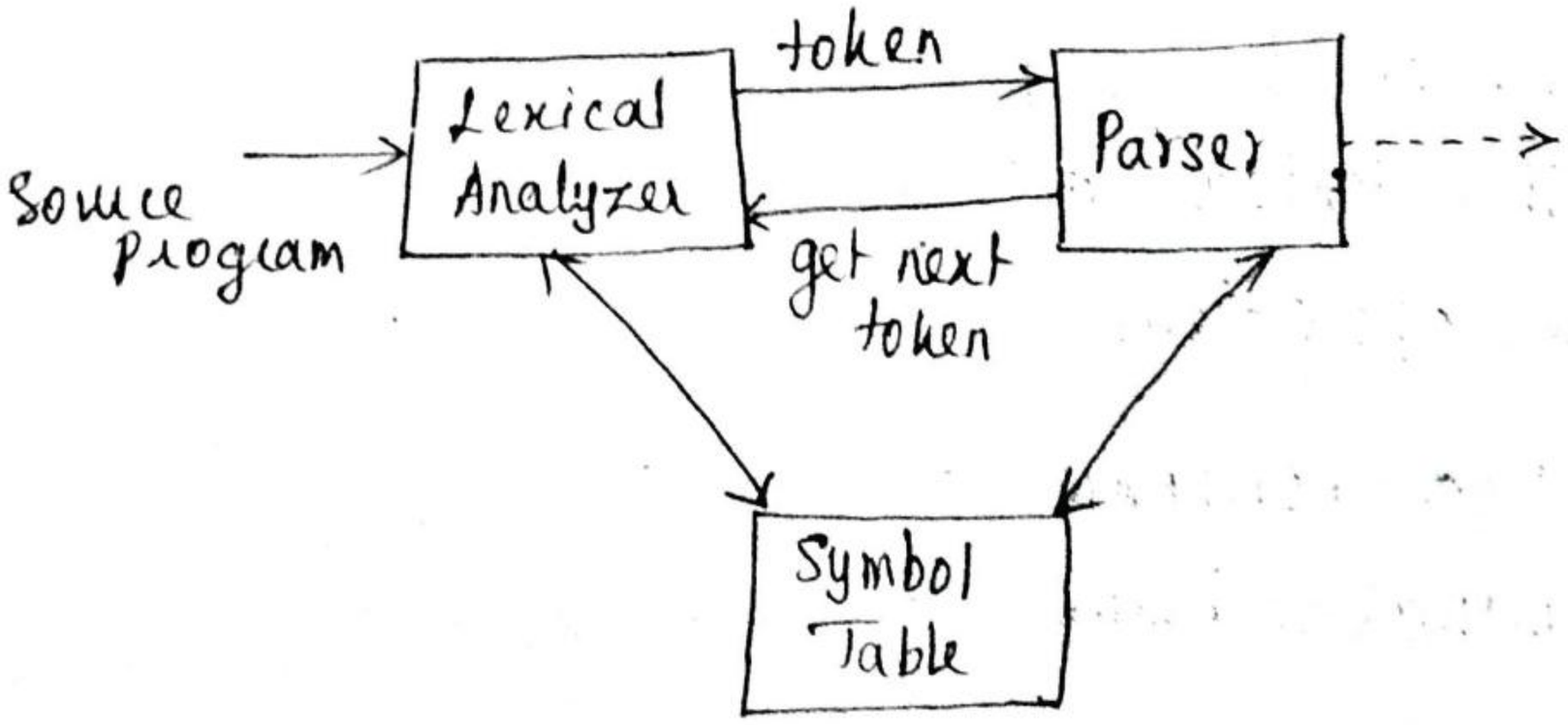
UNIT II

LEXICAL ANALYSIS

Need and Role of Lexical Analyzer - Lexical Errors - Expressing Tokens by Regular Expressions - Converting 1/12/18
Regular Expression to DFA - Minimization of DFA - Language for specifying Lexical Analyzer - LEX - Design of Lexical Analyzer for a Sample Language.

NEED AND ROLE OF LEXICAL ANALYZER:

* The lexical analysis 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.



* Upon receiving a 'get next token' command from the parser, the lexical analyzer reads input characters until it can identify the next token
* Lexical analyzer also perform certain secondary tasks at the user interface
  One such task is skipping out from the source program comments and white space in the form of blank, tab and new line characters
  → Another is correlating the error messages from the compiler with the source program
* Lexical analyzers are divided into a cascade of two phases
  1. Scanning
  2. Lexical Analysis
* The scanner is responsible for doing simple tasks while the lexical analyzer proper does the more complex operations.
* For eg) a FORTRAN compiler might use a scanner to eliminate blanks from the input.

Issues in Lexical Analysis

* There are several reasons for separating the analysis phase of compiling into lexical analyses and parsing

1 The separation of lexical analysis from syntax analyses often allows us to <u>simplify</u> One or the other of these phases
For eg) a parser embodying the conventions for comments and white space is significantly more complex than one that can assume comments and white space have already been removed by a lexical analyzer.

2. Compiler <u>efficiency</u> is improved.
* A separate lexical analyzer allows us to construct a specialized and potentially more efficient for the task
* Specialized buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.

3. Compiler <u>portability</u> is enhanced Input alphabet peculauities and other device specific anomolies can be restricted to the lexical analyzer.

INPUT BUFFERING!

* There are three general approaches to the implementation of a lexical Analyzer.

1. Use a lexical analyzer generator such as the Lex compiler to produce the lexical analyzer from a regular expression based specification In this case the generator provides routines for reading and buffering the input.

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

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

Buffer Pairs:

* For many source language there are times when the lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced.

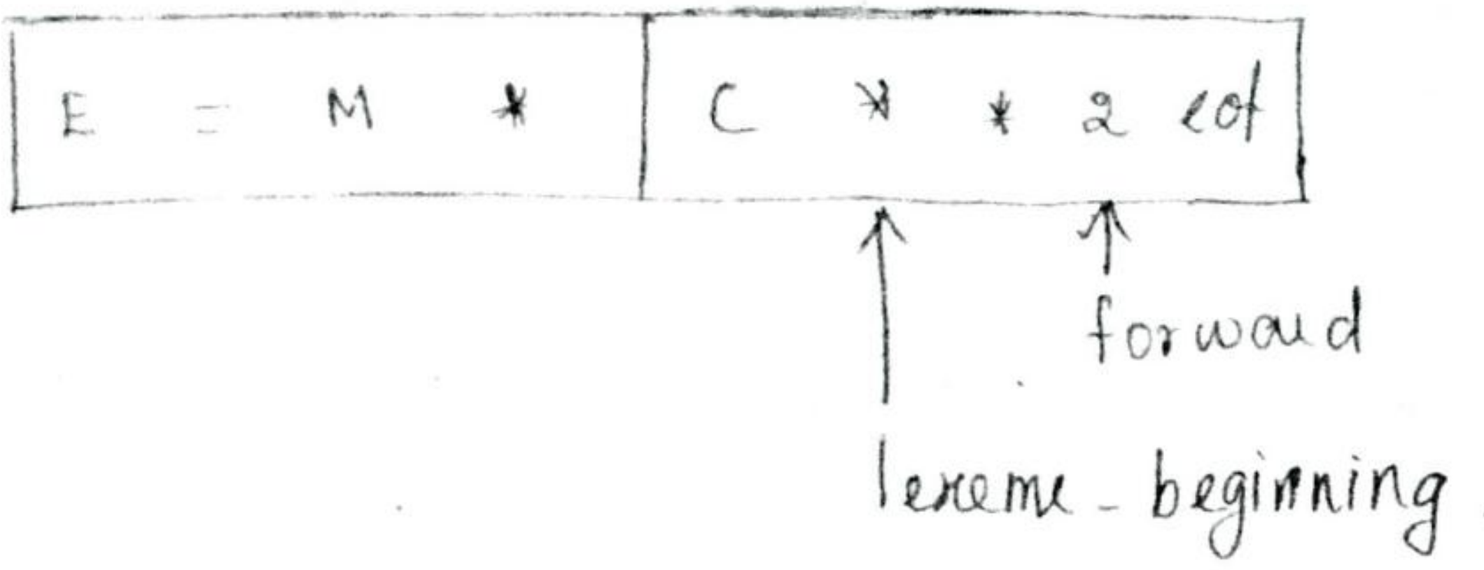
* Because a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

* Many buffering schemes can be used but the techniques are dependent on system parameters

* We use a buffer divided into two N-character halves.

N - is the no. of characters on one disk block

eg. 1024 or 4096.



* We read N input characters into each half of the buffer with one system read command rather than involving a read command for each input characters.
* If fewer than N characters remain in the input, then a special character eof is read into the buffer, after the input character
* Two pointers to the input buffer are maintained
    1. Forward pointer
    2. Lexeme pointer.
* Forward pointer scans ahead until a match for a pattern is found.
* The string of characters between the two pointers is the <u>current lexeme</u>.
* Initially, both pointers points to the first character of the next lexeme to be found.