Module I –Part II
Lexical Analysis

Text books

1. Compilers – Principles, Techniques & Tools , Aho, Ravi Sethi, D. Ullman

- **Introduction to compilers and lexical analysis:- Analysis of the source program - Analysis and synthesis phases, Phases of a compiler.**
- **Compiler writing tools. Bootstrapping.**
- **Lexical Analysis - Role of Lexical Analyser, Input Buffering, Specification of Tokens, Recognition of Tokens**

Lexical Analysis

- Deals with techniques for specifying and implementing lexical analyzers
- A simple way to build a lexical analyzer is to
  - Construct a diagram that illustrates the structure of the tokens for the source language
  - Hand translate the diagram into a program for finding tokens
- Efficient analyzers can be produced in this manner

The role of the lexical analyzer

- First phase of the compiler
- Main task – To read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis
- Schematic in fig 3.1,commonly implemented by making the lexical analyzer a subroutine or a co-routine of the parser

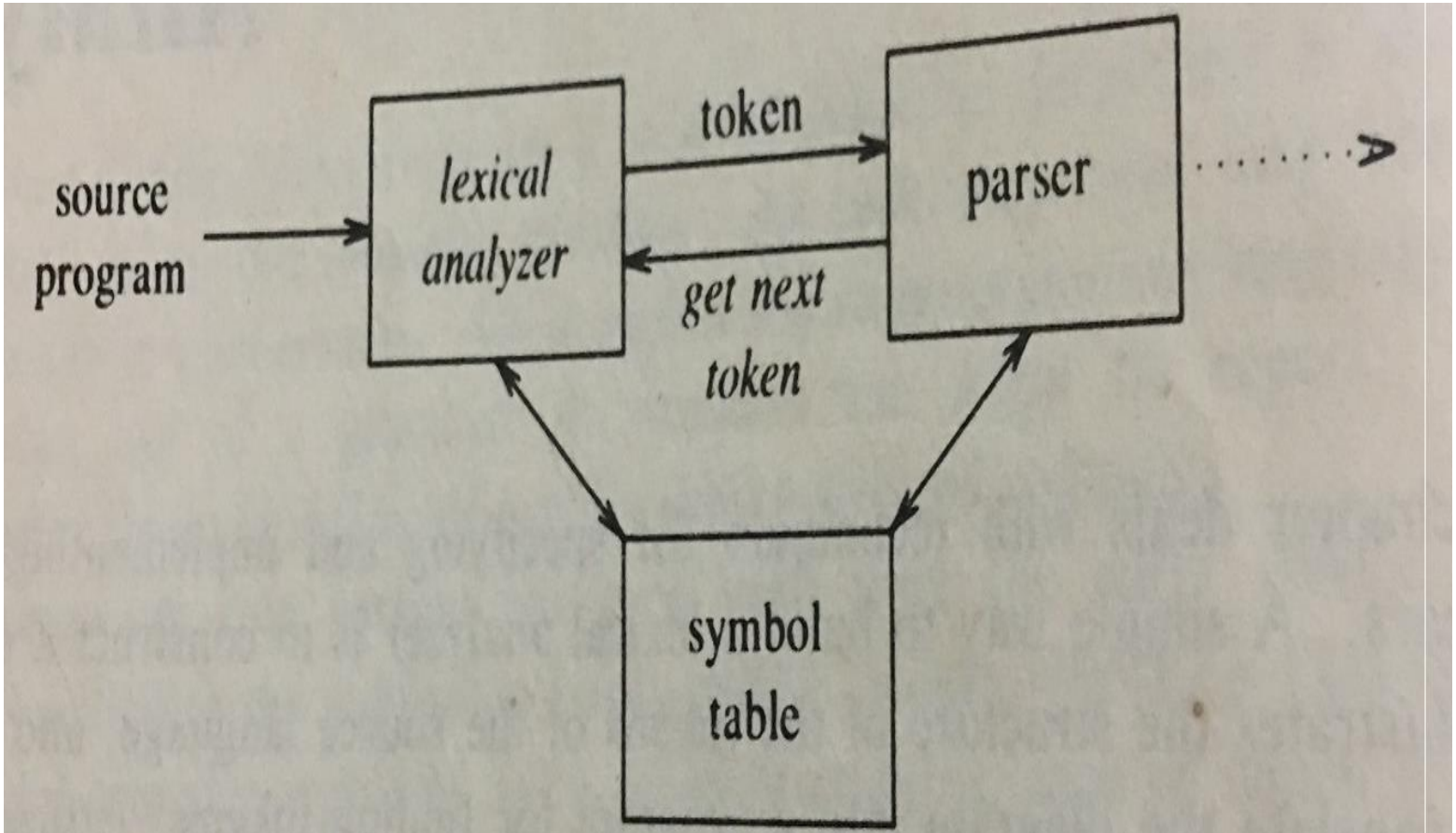

Fig. 3.1. Interaction of lexical analyzer with parser.

- Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token
- Since it reads the source text, it may also perform certain secondary tasks at the user interface
  - Stripping out comments and white spaces (blank, tab, newline)
  - Correlating error messages from the compiler with the source program
    - May keep track of the number of newline characters, so that a line number can be associated with an error message

- 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 the lexical analyzer proper does the more complex operations

Issues in Lexical Analysis

- Reasons for separating the analysis phase of compiling into lexical analysis and parsing

1. Simpler design
   - Separation allows to simplify one or other of these phases
   - For eg: a parser with code for handling white spaces and comments is more complex than one that assumes that they are removed by the lexical analyzer
   - If we are designing a new language, separating the lexical and syntactic conventions can lead to a cleaner overall language design

2 Improved compiler efficiency

- A separate lexical analyzer allows us to construct a specialized and potentially more efficient processor for the task
- A large amount of time is spent reading the source program and partitioning it into tokens
- Specialized buffering techniques for reading the characters and processing tokens can significantly speed up the performance of a compiler

3. Enhanced compiler portability

- 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 the lexical analyzer

Tokens, patterns and lexemes

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

Fig. 3.2. Examples of tokens.

- There is a set of strings in the input for which the same token is produced as output
- The set of strings is described by a rule called a “pattern” associated with the token
- The pattern is said to match each string in the set
- A lexeme is a sequence of characters in the source program that is matched by the pattern for a token
- For eg : In the Pascal statement

  const pi = 3.1416

  the substring pi is the lexeme for the token “identifier”

- Tokens are terminal symbols in the grammar for the source language
- Boldface names are used to represent tokens
- The lexemes matched by the pattern for the token represent strings of characters in the source program that can be treated together as a lexical unit
- In most programming languages, the following constructs are treated as tokens :

  keywords, operators, identifiers, constants, literal strings and punctuation symbols such as parentheses, commas and semicolons
- In the above eg, when the character sequence pi appears in the source program, a token representing an identifier is returned to the parser

- A pattern is a rule describing the set of lexemes that can represent a particular token in source program
- The pattern for the token **const** in the eg, is just the single string “const” that spells out the keyword
- The pattern for the token **relation** is the set of all Pascal relational operators
- To describe precisely the patterns for more complex tokens like **id** and **num**,we use the following regular expression

letter -> A|B|.....|Z|a|b|.....|c

digit -> 0|1|.....|9

id -> letter (letter | digit)*

digits -> digit digit*

num -> digits

Attributes for Tokens

- When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler
- For eg, the pattern num matches both the strings 0 and 1, but the code generator must know what string was actually matched
- The lexical analyzer collects information about tokens into their associated attributes
- These attributes are stored in the symbol table

- The tokens and associated attribute-values for the Fortran statement

E = M * C * * 2

<id, pointer to the symbol table entry for E>

<assign_op>

<id, pointer to symbol table entry for M>

<multi_op>

<id, pointer to symbol table entry for C>

<exp_op>

<num,integer value 2>

Lexical Errors

- The lexical analyzer alone will not be able to identify the errors in the source program
- If the string fi is encountered in a C program for the first time in the statement fi (a==f(x)) .....,a lexical analyzer wont be able to tell whether fi is a misspelling of the keyword if or an undeclared function identifier
- In some situation, the lexical analyzer will be unable to proceed because none of the patterns matches a prefix of the remaining input
- There are different methods for the lexical analyzer for recovering from such situations

1. Panic mode recovery
   - Delete successive characters from the remaining input until the lexical analyzer can find a well-formed token
2. Other possible error-recovery actions are
   a. Deleting an extraneous character
   b. Inserting a missing character
   c. Replacing an incorrect character by a correct character
   d. Transposing two adjacent characters

Implementation of a lexical analyzer

- Three general approaches

1. Use a lexical analyzer generator such as Lex to produce the lexical analyzer from a regular expression based specification. Here the generator provides 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 the input

Input buffer pairs

- Helps the lexical analyzer to look ahead several characters beyond the lexeme for a pattern before a match can be announced
- Many buffering schemes are available
- Here we use a buffer divided into two N-character halves as in fig 3.3
- Typically, N is the number of characters on one disk block

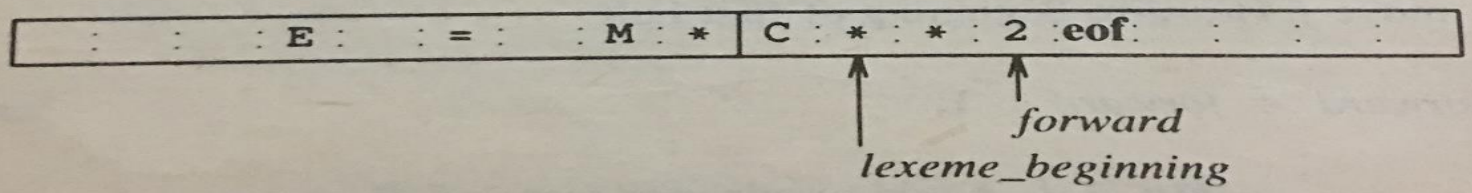

Fig. 3.3. An input buffer in two halves.

- N input characters are read into each half of the buffer with one system read command, rather than invoking a read command for each input character
- If fewer than N characters remain in the input, then a special character eof is read into the buffer after the input characters
- Ie, eof marks the end of the source file and is different from any input character

Method

- Two pointers to the input buffers
- The 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
- The “forward pointer” scans ahead until a match for a pattern is found
- Once the lexeme is found, the forward pointer is set to the character at its right end
- After the lexeme is processed, both pointers are set to the character immediately past the lexeme
- Here white spaces and comments are treated as patterns yielding no tokens

- When the forward pointer move past the half way mark, the right half is filled with new input characters
- When the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer

- Works well most of the time
- But, the amount of look ahead is limited

```
if forward at end of first half then begin
        reload second half;
        forward = forward +1
end
else if forward at end of second half then begin
        reload first half;
        move forward to the beginning of the first half
end
else
        forward := forward + 1
```

Fig 3.4 Code to advance forward pointer

- Here each time we need to check the position of the forward pointer
- These two tests can be reduced to one if we extend each buffer half to hold a sentinel character at the end
- The sentinel is a special character that cannot be part of the source program
- A natural choice is eof

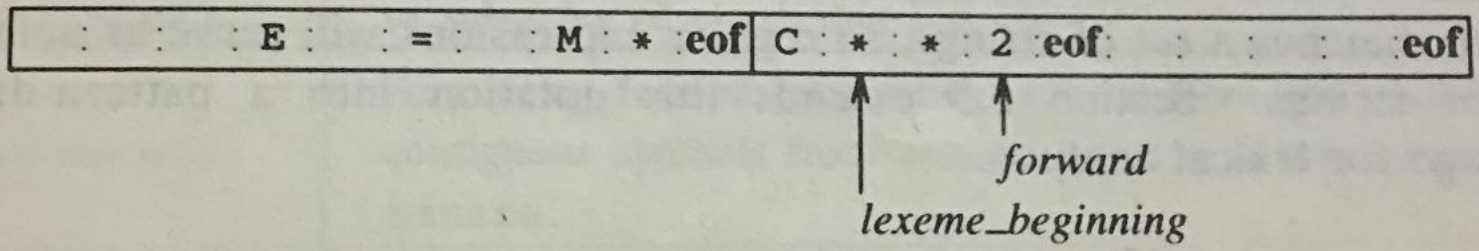

Fig. 3.5. Sentinels at end of each buffer half.

```
forward = forward +1
if forward↑ = eof then begin
   if forward at end of first half then begin
         reload second half;
         forward = forward +1
   end
   else if forward at end of second half then begin
         reload first half;
         move forward to the beginning of the first half
   end
   else
         terminate lexical analysis
end
```

Fig 3.6 Look ahead code with sentinels

- With the arrangement of fig 3.5 we can use the code shown in fig 3.6 to advance the forward pointer
- Here most of the code perform only one test to see whether forward points to an eof
- Only when we reach the end of a buffer half or the end of the file do we perform more tests
- Since N input characters are encountered between eof's, the average number of tests per input character is very close to 1

Specification of tokens

- Regular expressions are an important notation for specifying patterns
- Each pattern matches a set of strings, so regular expressions will serve as names of sets of strings

Strings and Languages

- Alphabet or character class
  - Any finite set of symbols
  - Eg : letters or characters
  - The set {0,1} is the binary alphabet
  - ASCII and EBCDIC are computer alphabets
- String
  - A string over some alphabet is a finite sequence of symbols drawn from that alphabet
  - Sentence and word are synonyms
  - Length of a string |s|, the number of occurrences of symbols in s
  - The empty string ε is a special string of length 0
- Common terms associated with parts of strings are:

TERM	DEFINITION
prefix of *s*	A string obtained by removing zero or more trailing symbols of string *s*; e.g., ban is a prefix of banana.
suffix of *s*	A string formed by deleting zero or more of the leading symbols of *s*; e.g., nana is a suffix of banana.
substring of *s*	A string obtained by deleting a prefix and a suffix from *s*; e.g., nan is a substring of banana. Every prefix and every suffix of *s* is a substring of *s*, but not every substring of *s* is a prefix or a suffix of *s*. For every string *s*, both *s* and ϵ are prefixes, suffixes, and substrings of *s*.
proper prefix, suffix, or substring of *s*	Any nonempty string *x* that is, respectively, a prefix, suffix, or substring of *s* such that $s \neq x$.
subsequence of *s*	Any string formed by deleting zero or more not necessarily contiguous symbols from *s*; e.g., baaa is a subsequence of banana.

Fig. 3.7. Terms for parts of a string.

- Language
  - Any set of strings over some fixed alphabet
- If x and y are strings, then the concatenation of x and y written xy, is the string formed by appending y to x
- Eg : If x = dog and y =house, then xy = doghouse
- The empty string is the identity element under concatenation
- If concatenation is a “product”, then string “exponentiation” can be defined as follows:
  - Define s^0 to be ε
  - For i>0, define $s^i = s^{i-1} s$
  - Since $\varepsilon s = s$, $s^1 = s$, $s^2 = ss$, $s^3 = sss$ etc

Operation on languages

- Union, Concatenation and closure

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L and M written LM	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ L^+ denotes "one or more concatenations of" L.

Fig. 3.8. Definitions of operations on languages.

- We can also generalize the “exponentiation” operator to languages by defining L^0 to be {ε} and L^1 to be $L^{i-1}L$
- Thus L^i is L concatenated with itself i-1 times

- Eg : Let L be the set {A,B,…Z,a,b,…z} and D the set {0,1,,…9}

  So L is the alphabet consisting of the set of upper and lower case letters and D is the alphabet consisting of the set of the ten decimal digits

  since, a symbol can be regarded as a string of length one, the sets L and D are finite languages

New languages created from L and D applying operators

1. L U D is the set of letters and digits
2. LD is the set of strings consisting of a letter followed by a digit
3. $L^4$is the set of all four-letter strings
4. L* is the set of all strings of letters, including ε, the empty string
5. L(L U D)* is the set of all strings of letters and digits beginning with a letter
6. D+ is the set of all strings of one or more digits