**CHAPTER – 2**

**LEXICAL ANALYSIS**

**3.1 THE ROLE OF THE LEXICAL ANALYZER**

**Main task**

* The lexical analyzer 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.



**Secondary tasks**

* One such task is stripping out from the source program comments and white space in the form of blank, tab, and newline characters.
* Another is correlating error messages from the compiler with the source program. For example, the lexical analyzer may keep track of the number of newline characters seen, so that a line number can be associated with an error message.

**3.2 ISSUES IN LEXICAL ANALYSIS**

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

* **Simpler design** - The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases. For example, 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.
* **Compiler efficiency is improved** - 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. Special buffering techniques for reading input characters and processing tokens can significantly speed up the performance of a compiler.
* **Compiler portability is enhanced** - Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer. The representation of special or non-standard symbols, such as in Pascal, can be isolated in the lexical analyzer.

**3.3 TOKENS, PATTERNS, LEXEMES**

**Pattern** - There is a set of strings in the input for which the same token is produced as output. This 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.

Eg. The pattern for the identifier token id, is

**id-> letter(letter/digit)\***

**Lexeme -** A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

For example, in the Pascal statement,

**const pi = 3.1416**

the substring pi is a lexeme for the token "identifier".



**Token** – A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

**Example of tokens:** key words, operators, identifiers, constants, literal strings, and punctuation symbols such as parentheses, commas, and semicolons.

**3.4 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.
* The lexical analyzer collects information about tokens into their associated attributes.

**Example**

The tokens and associated attribute-values for the FORTRAN statement **E=M \* C\*\* 2** are written below as a sequence of pairs:

**<id, pointer to symbol-table entry for E>**

**<assign\_op,>**

**<id, pointer to symbol-table entry for M>**

**<mult\_op,>**

**<id, pointer to symbol-table entry for C>**

**<exp\_op,>**

**<num, integer value 2>**

**3.5 LEXICAL ERROR**

* Few errors are discernible at the lexical level alone, because a lexical analyzer has a very localized view of a source program.
* If the string f i is encountered in a C program for the first time in the context

fi(a==b)

a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier.

* Since fi is a valid identifier, the lexical analyzer must return the token for an identifier and let some other phase of the compiler handle any error.
* suppose a situation does arise in which the lexical analyzer is unable to proceed because none of the patterns for tokens matches a prefix of the remaining input.
* "panic mode" recovery will be taken that delete successive characters from the remaining input until the lexical analyzer can find a well-formed token.
* Other possible error-recovery actions are:

1. deleting an extraneous character

2. inserting a missing character

3. replacing an incorrect character by a correct character

4. transposing two adjacent characters.

**3.6 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 for sets of strings.

**Strings and Languages**

* The term **alphabet** or character class denotes any finite set of symbols.
* Typical examples of **symbols** are letters and characters. The set {0,1} is the binary alphabet. ASCII and EBCDIC are two examples of computer alphabets.
* A **string** over some alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms sentence and word are often used as synonyms for the term "string."
* The length of a string s, usually written |s|, is the number of occurrences of symbols in s. For example, banana is a string of length six.
* The empty string denoted, is a special string of length zero.
* If x and y are strings, then the concatenation of x and y, written xy, is the string formed by appending y to x, for example, if x = dog and y = house, then xy = doghouse. The empty string is the identity element under concatenation.
* That is, s = s = s.
* **exponentiation** of the string, s**0** to be , and for i >O define si to be si -1 s. Since s is s itself, s' = s. Then, s2 = ss, s3 = sss, and so on.



**Language**

The term Language denotes any set of strings over some fixed alphabet.

**Operation on languages**

There are several important operations that can be applied to languages. For lexical analysis, we are interested primarily in **union, concatenation, and closure**.

**Example**, Let L is the set {A, B. . . Z, a, b,. . , z} and D the set (0, 1. . . 9).

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. L4 is the set of all four-letter strings.

4. L\* is the set of all strings of letters, including, the empty string.

5. L (L U D)\* is the set of all string of letters and digits beginning with a letter.

6. D+ is the set of all strings of one or more digits.

**Regular Expressions**

* Regular expressions are an important notation for specifying patterns and sets.
* With this notation, we might define Pascal identifiers as L\* denotes "zero or more instance of”' parenthesized expression.

**letter(letter|digit)\***

* The vertical bar here means "or," the parentheses are used to group sub expressions, the star means "zero a more instances of" the parenthesized expression.
* A regular expression is built up out of simpler regular expressions using a set of defining rules. Each regular expression r denotes a language L(r).

Here are the rules that define the regular expressions over alphabet **∑**.

1. € is a regular expression that denotes {€}, that is, the set containing the empty string.

2. If a is a symbol in **∑**, then a is a regular expression that denotes {a}, i.e., the set containing the string a.

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

a) (r) | (s) is a regular expression denoting L(r)U L(s).

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

C) (r)\* is a regular expression denoting (L(r))\*.

* A language denoted by a regular expression is said to be a regular set.
* Unnecessary parentheses can be avoided in regular expressions if we adopt the conventions that:

1 . the unary operator \* has the highest precedence and is left associative,

2. concatenation has the second highest precedence and is left associative,

3. | has the lowest precedence and is left associative.

Under these conventions, **(a) | ((b)\*(c))** is equivalent to **a | b\*c**.

**Example**, Let ∑ = {a, b}

1. The regular expression a | b denotes the set {a, b}.

2. The regular expression (a | b) (a | b) denotes {aa, ab,ba, bb} the set of all strings of a's and b's of length two.

3. The regular expression a\* denotes the set of all strings of zero or more a's, i.e., {€, a, aa, aaa,….).

**Regular definition**

* Regular definition is names given to regular expressions.
* lf ∑ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

****

* where each di is a distinct name, and each ri is a regular expression over the symbols in ∑ U {d1, d2 , . . , di-1}, i.e., the basic symbols and the previously defined names.
* Here is a regular definition for this set.



**Notational Short hands**

1. One or more instances . The unary postfix operator + means "one or more Instances of." If r is a regular expression that denotes the language L(r) , then (r)+ is a regular expression that denotes the language (L (r))+ .
2. Zero or one instances. The unary postfix operator ? means "zero or one insnstance of." The notation r? is a shorthand for r | €.
3. Character classes. The notation [abc] where a. b, and c are alphabet symbols denotes the regular expression a | b | c. An abbreviated character class such as [a-z] denotes the regular expression a | b | . . . | z.

**3.7 RECOGNITION OF TOKENS**

It addresses the question of how to recognize them.



* For this language fragment the lexical analyzer will recognize the keywords **if, then, else,** as well as the lexemes denoted by **relop, id,** and **num.**
* In addition, we assume lexemes are separated by white space, consisting of nonnull sequences of blanks. tabs, and newlines. Our lexical analyzer will strip out white space.

**ws 🡪 (blank | tab | newline)+**

* If a match for ws is found the lexical analyzer does not return a token to the parser. Rather, it proceeds to find a token following the white space and returns that to the parser.
* Our goal is to construct a lexical analyzer that will isolate the lexeme for the next token in the input buffer and produce as output a pair consisting of the appropriate token and attribute-value, wing the translation table given in Fig. 3.10.



**Transition diagram**

* An intermediate step of the construction of a lexical analyzer, in a stylized flowchart, called a **transition diagram**.
* **Transition diagrams** depict the actions that take place when a lexical analyzer is called by the parser to get the next token.
* We use a transition diagram to keep track of information about characters that are seen as the forward pointer scans the input.
* Positions in a transition diagram are drawn as circles and are called **states**. The states are connected by arrows, called **edges**. Edges leaving state s have labels indicating the input characters that can next appear after the transition diagram has reached state S. The label other refers to any character that is not indicated by any of the other edges leaving s.
* One state is labelled the **start** state; it is the initial state of the transition diagram where control resides when we begin to recognize a token.
* Figure 3.11 shows a transition diagram for the patterns **>= and >.** The transition diagram works as follows. Its start state is state 0. In state 0, we read the next input character. The edge labelled > from state 0 is to be followed to state 6 if this input character is >. Otherwise, we have failed to recognize either > or >=.



* Notice that the character > and another extra character are read as we follow the sequence of edge from the start state to the accepting state 8. Since the extra character is not a part of the relational operator >, we must retract the forward pointer one character. We use a **\*** to indicate states on which this input retraction must take place.

**Example, A transition diagram for the token relop is shown in Fig. 3.12.**

****