1. *Lexical Analysis Phase*

The purpose of the lexical analyzer is to read the source program, one character at a time, and to translate it into a sequence of primitive units called tokens. Keywords, identifiers, constants, and operators are examples of tokens.

Here we discuss the problem of designing and implementing lexical analyzers. We shall see that there are several aspects to this problem. First, we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notion that can be used to describe essentially all the tokens of programming languages.

Second, having decided what the tokens are, we need some mechanism to recognize these tokens, in the input stream. We can see that transition diagram and finite automata are convenient ways of designing token recognizers.

- **The Role of the Lexical Analyzer**

  The lexical Analyzer could be a separate pass, placing its output on an intermediate file from which the parser would then take its input, or the lexical analyzer and parser are together in the same pass where the lexical analyzer acts as a subroutine which is called by the parser whenever it needs a new token. This organization eliminates the need for the intermediate file. In this arrangement, the lexical analyzer returns to the parser a representation for the token it has found. The representation is an integer code if the token is a simple construct such as a left parenthesis, comma or colon. The representation is a pair consisting of an integer code and a pointer to a table if the token is a more complex element such as an identifier or constant. The
integer code gives the token type and the pointer points to the value of that token. Pairs are also returned whenever we wish to distinguish between instances of tokens. For example, we may treat "operator" as a token and let the second component of the pair indicate whether the operator found is +, *, and so on.

• The Need for Lexical Analyzer

The purpose of splitting the analysis of the source program into two phases, lexical analysis and syntax analysis is to simplify the overall design of a compiler. It is easier to specify the structures of tokens than the syntactic structure of the source program. Consequently, we can construct a more specialized, and hence more efficient, recognizer for tokens than for the syntactic structure.

Other functions sometimes performed by the lexical analyzer are keeping track of line numbers, stripping out white space (such as redundant blanks and tabs), and deleting comments.

Specification of tokens

String and languages

First we introduce some terms that dealing with languages. We shall use the term "alphabet" or "character class", to denote any finite set of symbols.

The set \{0, 1\} is an alphabet. It consist of the two symbols 0 and 1, and it is called binary alphabet. Two important examples of programming language alphabets are the ASCII and EBCDIC character sets.

A string is a finite sequence of symbols, such as 0011. Sequence and word are synonyms for string. The length of the string x, usually denoted |x| is the
total number of the symbols in x. For example 01101 is a string of length 5. A
special string in the empty string, which we shall denote by \( \epsilon \). This string is of
length zero.

If x and y are string, then the concatenation of x and y; written x.y or xy, is the
string formed by the symbol of x followed by the symbol of y. For example if
x=abc and y=de, where a, b, c, d, e are symbols, then xy=abcde. The condition
of the empty string with any string is that string, more formally \( \epsilon x=x \epsilon =x \).

We may think of condition as a "product". It thus makes sense to talk of
exponentiation of string as representing an iterated project. For example, \( x^1=x \),
\( x^2=xx \), \( x^3=xxx \) and so on. In general, \( x^i \) is the string x repeated i times. As a
useful convention, we take \( x^0 \) to be \( \epsilon \) for any string x. Thus, \( \epsilon \), is the identity
of concatenation.

The term languages to mean any set of string formed from some specific
alphabet.

The notation of concatenation can also be applied to languages. If L and M are languages, then L.M is the language consisting of all string xy, which
can be found by selecting a string x from L, and a string y from M, and
concatenating them in that order. That is,

\[ LM= \{xy|x \text{ is in L and } y \text{ in M}\} \]

we call LM the concatenation of L and M.

**Example:** Let L be \( \{0, 01, 110\} \), and let M be \( \{10, 110\} \).

Then \( LM= \{010, 0110, 01110, 11010, 110110\} \).

We use \( L^i \) to stand for \( LL \ldots L \) (i times). It is logical to define \( L^0 \) to be \( \{ \epsilon \} \).
The union of languages $L$ and $M$ is given by

$$L \cup M = \{x \mid x \text{ is in } L \text{ or } x \text{ is in } M\}.$$ 

The empty set, $\emptyset$, is the identity under union, since

$$\emptyset \cup L = L \cup \emptyset = L$$

And

$$\emptyset L = L \emptyset = \emptyset$$

There is another operation on languages which plays an important role in specifying tokens. This is the kleenclosure operator. We use $L^*$ to denote the concatenation of language $L$ with itself any number of times.

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

**Example**

Let $D$ be the language consisting of the string 0, 1... 9, that is, each string is a single decimal digit. Then $D^*$ is all strings of digits, including the empty string. For example, if $L = \{aa\}$, then $L^*$ is all string of an even number of a's, since $L^0 = \{\varepsilon\}$, $L^1 = \{aa\}$, $L^2 = \{aaaa\}$, ... . If we wished to exclude $\varepsilon$, we could write $L.(L^*)$, to denote that language. That is:-

$$ L.(L^*) = L.\bigcup_{i=0}^{\infty} L^i = \bigcup_{i=0}^{\infty} L^{i+1} = \bigcup_{i=1}^{\infty} L^i $$

We shall often use the $L^*$ for $L.(L^*)$. The unary postfix operator $+$ is called positive closure, and denotes "one or more instances of".

**A simple Approach to the Design of Lexical Analyzers**

One way to begin the design of any program is to describe the behavior of the program by a flowchart. This approach is particularly useful when the program is a lexical analyzer, because the action taken is highly dependent on what characters have been seen recently. Remembering previous characters by
the position in a flowchart is a valuable tool, so much so that a specialized kind of flowchart for lexical analyzer, called a transition diagram, has evolved. In a transition diagram, the boxes of the flowchart are drawn as circles and called states. The states are connected by arrow, called edges. The labels on the various edges leaving a state indicate the input characters that can appear after that state.

Fig (4) Transition diagram for identifier

Fig (4) shows a transition diagram for an identifier, defined to be a letter followed by any number of letters or digits. The starting state of the transition diagram is state 0, the edge from which indicates that the first input character must be a letter. If this is the case, we enter state 1 and look at the next input character if this is a letter or the digit, we continue this way, reading letters and digits, and making transition from state 1 to itself, until the next input characters is a delimiter for an identifier, which we have assume is any character that is not a letter or a digit. On reading the delimiter, we enter state 2.

To turn a collection of transition diagram into a program, we construct a segment of code for each state. The first step to be done in the code for any state is to obtain the next character from the input buffer. For this purpose we use a function GETCHAR, which returns the next character, advancing the look ahead pointer at each call. The next step is to determine which edge, if any, out of the state is labeled by a character or class of characters that includes the character just read. If such an edge is found, control is transferred
to the state pointed to by that edge. If no such edge is found, and the state is not one which indicated that a token has been found (indicated by a double circle), we have fail to find this token. The look ahead pointer must be retracted to where the beginning pointer is, and another token must be searched for, using another transition diagram. If all transition diagrams have been tried without success, a lexical error has been detected, and an error correction routine must be called.

Consider the transition diagram in fig (4), the code for state 0 might be:-

State 0: C:=GETCHAR ();
If LETTER(C) then goto state 1
else FAIL ()

Here, LETTER is a procedure which returns true if and only if C is a letter. Fail() is a routine which retracts the lookahead pointer and starts up the next transition diagram, if there is one, or calls the error routine. The code for state 1 is:

State 1 C:=GETCHAR ();
if LETTER (C) or DIGIT (C) then goto state 1
else if DELIMITER(C) then goto state 2
else FAIL ()

DIGIT is a procedure which returns true if and only if C is one of the digits 0, 1...9. DELIMITER is a procedure which returns true whenever C is a character that could follow an identifier. If we define a delimiter to be any character that is not letter or digit, then the clause "if DELIMITER (C) then", need not be presented in state 1. To detect errors more effectively we might define a delimiter precisely (e.g., blank, arithmetic or logical operator, left or
right parenthesis, equal sign, colon, semicolon, or comma), depending on the language being compiled.

State 2 indicates that an identifier has been found. Since the delimiter is not part of the identifier, we must retract the lookahead pointer one character, for which we use a procedure RETRACT. We use '*' to indicate states on which input retraction must take place. We must also install the newly-found identifier in the symbol table if it is not already there, using the procedure INSTALL*. In state 2 we return a pair consisting of the integer code for an identifier, which we denote by id, and a value that is a pointer to the symbol table returned by INSTALL. The code for state 2 is:

\[
\text{State 2: } \text{RETRACT ( )} \\
\text{return (id, INSTALL ( ))}
\]

If blank must be skipped in the language at hand, we should include in the code for state 2 a step that moved the beginning pointer to the next non-blank.

Fig (5) shows a list of tokens that we want to recognize using token recognizer that use transition diagram explained in fig (6).

<table>
<thead>
<tr>
<th>Token</th>
<th>Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>1</td>
<td>-------</td>
</tr>
<tr>
<td>end</td>
<td>2</td>
<td>-------</td>
</tr>
<tr>
<td>if</td>
<td>3</td>
<td>-------</td>
</tr>
<tr>
<td>then</td>
<td>4</td>
<td>-------</td>
</tr>
<tr>
<td>else</td>
<td>5</td>
<td>-------</td>
</tr>
<tr>
<td>identifier</td>
<td>6</td>
<td>Pointer to Symbol Table</td>
</tr>
<tr>
<td>constant</td>
<td>7</td>
<td>Pointer to Symbol Table</td>
</tr>
<tr>
<td>&lt;</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>&lt;=</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>=</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>&gt;</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>&gt;=</td>
<td>8</td>
<td>6</td>
</tr>
</tbody>
</table>

Fig (5) Token Recognizer
A more efficient program can be constructed from a single transition diagram than from a collection of diagrams, since there is no need to backtrack and rescan using a second transition diagram. In fig (6), we have combined all keywords into one transition diagram. However, if we attempt to combine the diagram for identifiers with that for keywords, difficulties arise. For example,
one seeing the three letters BEG, we could not tell whether to be in state 3 or state 24.

In fig (6), each keyword is treated as a separate token, whereas all relops are combine into one token class, with the associated token value distinguishing one relops from another.

Let us now consider an example if the action of the lexical analyzer constructed from the transition diagram of fig (6). On seeing IFA followed by a blank, the lexical analyzer would traverse state 0, 15, and 16, then fail and retract the input to I. It would then startup the second transition diagram at state 23, traverse state 24 three times, go to state 25 on the blank, retract the input one position, install IFA in the symbol table.