

## 1.Introduction

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called *compilers*.

### 1.1. Programming Languages

Hierarchy of Programming Languages based on increasing machine independence includes the following:

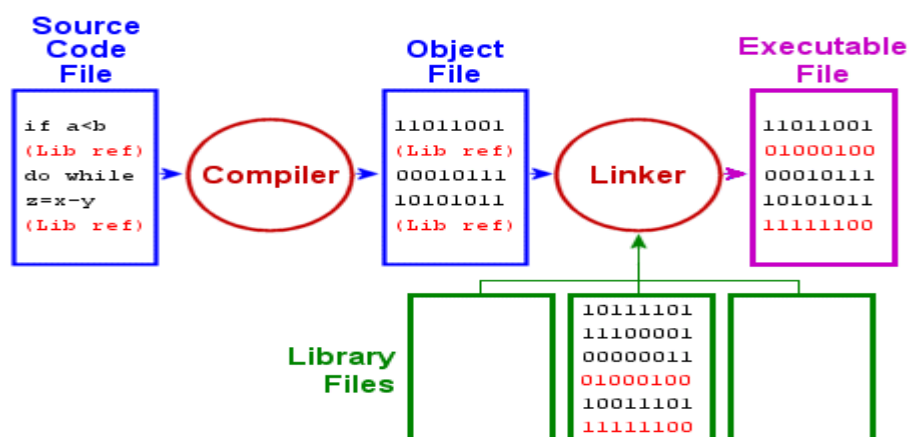
- 1- Machine – level languages.
- 2- Assembly languages.
- 3- High – level or user oriented languages.
- 4- Problem - oriented language.

1- Machine level language: is the lowest form of computer. Each instruction in program is represented by numeric code, and numerical addresses are used throughout the program to refer to memory location in the computer's memory.

2- Assembly language: is essentially symbolic version of machine level language, each operation code is given a symbolic code such ADD for addition and MULT for multiplication.

3- A high level language such as Pascal, C.

4- A problem oriented language provides for the expression of problems in specific application or problem area .examples of such as languages are SQL for database retrieval application problem oriented language.



### 1.1.1.Translator

A translator is program that takes as input a program written in a given programming language (the source program) and produce as output program in another language (the object or target program). As an important part of this translation process, the compiler reports to its user the presence of errors in the source program. If the source language being translated is assembly language, and the object program is machine language, the translator is called **Assembler**.

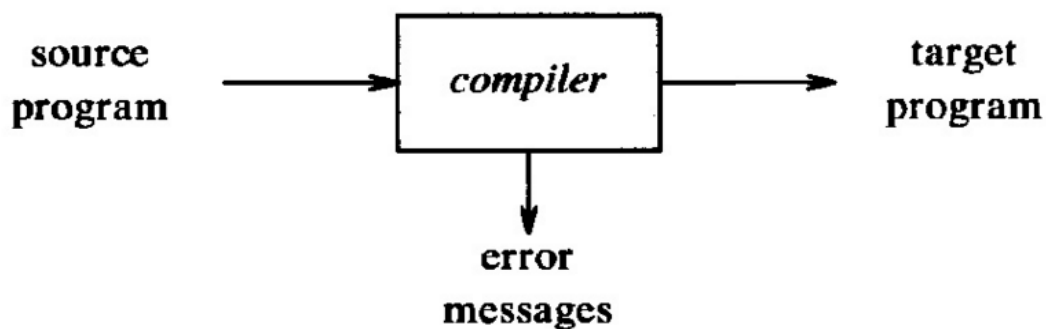


Fig (1)

A translator, which transforms a high level language such as C in to a particular computers machine or assembly language, called **Compiler**. Another kind of translator called an **Interpreter** process an internal form of the source program and data at the same time. That is interpretation of the internal source from occurs at run time and an object program is generated Fig (2) which illustrate the interpretation process.

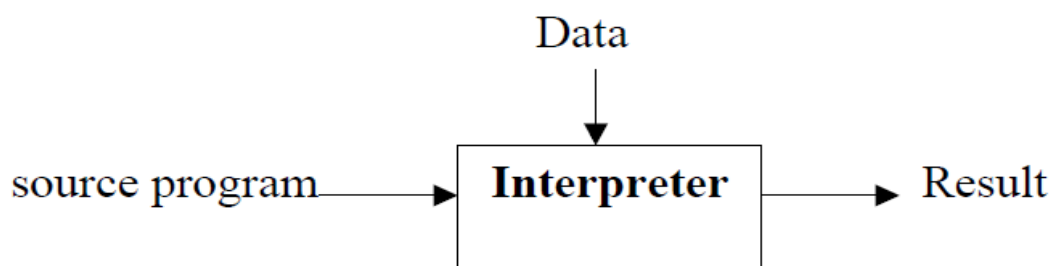


Fig (2)

### 1.1.2Compiler

Is a program (translator) that reads a program written in one language, (the source language) and translates into an equivalent program in another language (the target language).

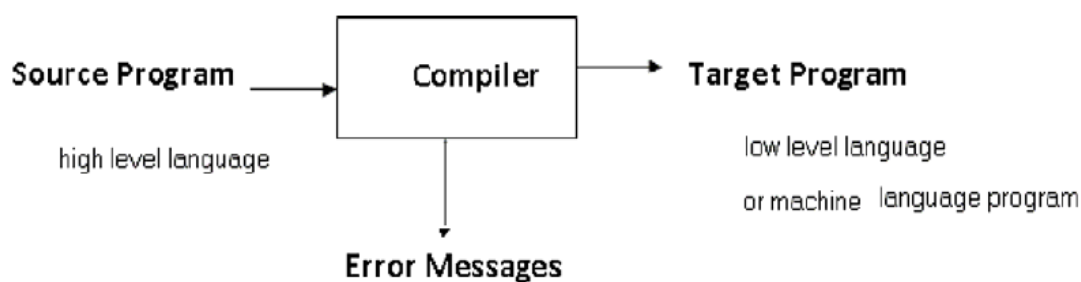


Fig (3)

The time at which the conversion of the source program to an object program occurs is called (compile time) the object program is executed at (run time). Fig (4) illustrate the compilation process Note that the program and data are processed at different times, compile time and run time respectively.

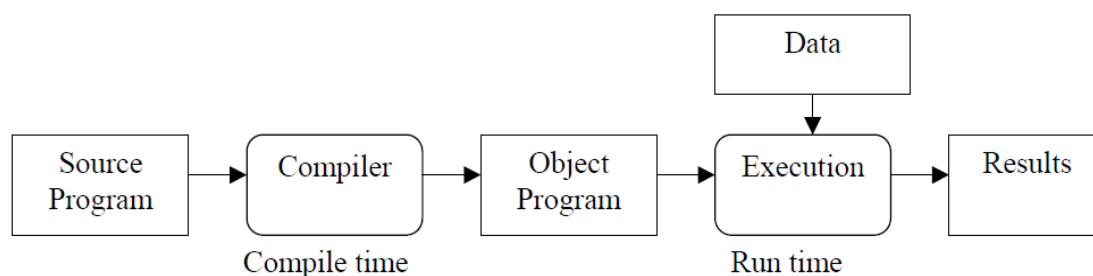
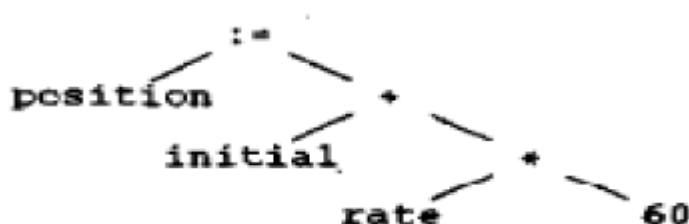


Fig (4) Compilation process

### 1.1.3. Analysis - Synthesis model of compilation

There are two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation. During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree. Often, a special kind of tree called a syntax tree is used, in which each node represents an operation and the children of a node represent the arguments of the operation. For example, a syntax tree for an assignment statement is shown below



syntax tree for position :- initial + rate \*60

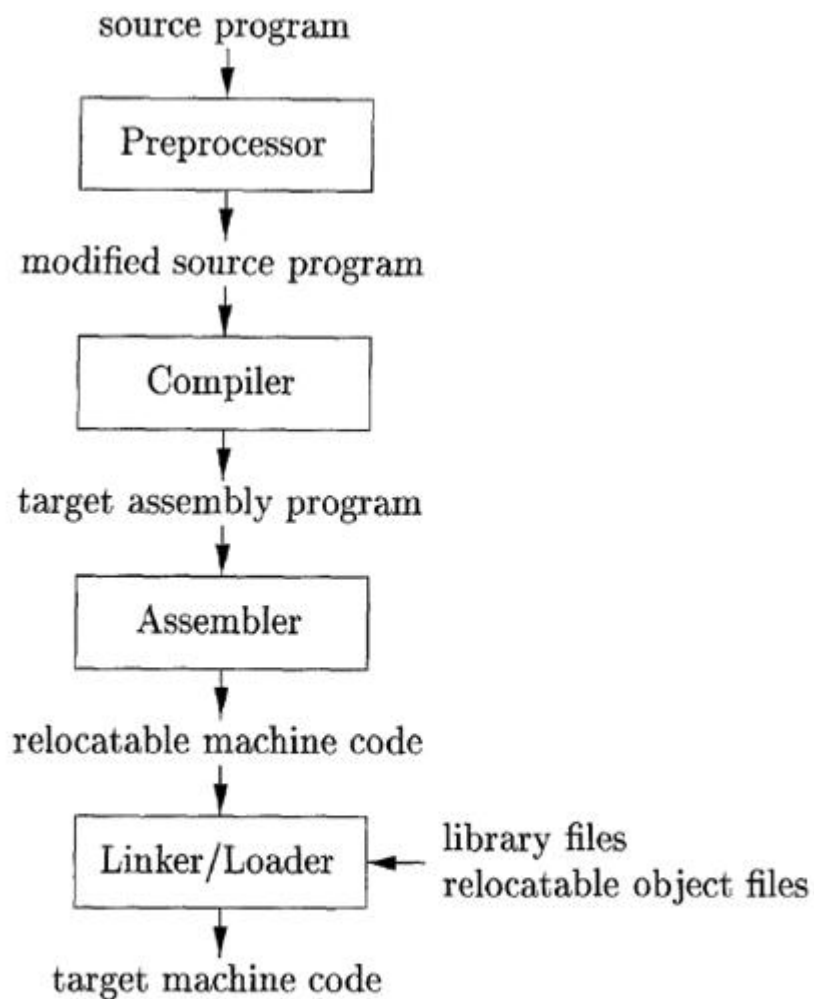


Figure 5 A language-processing system

## 2. The Structure of a Compiler

Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end. If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in Fig.6. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

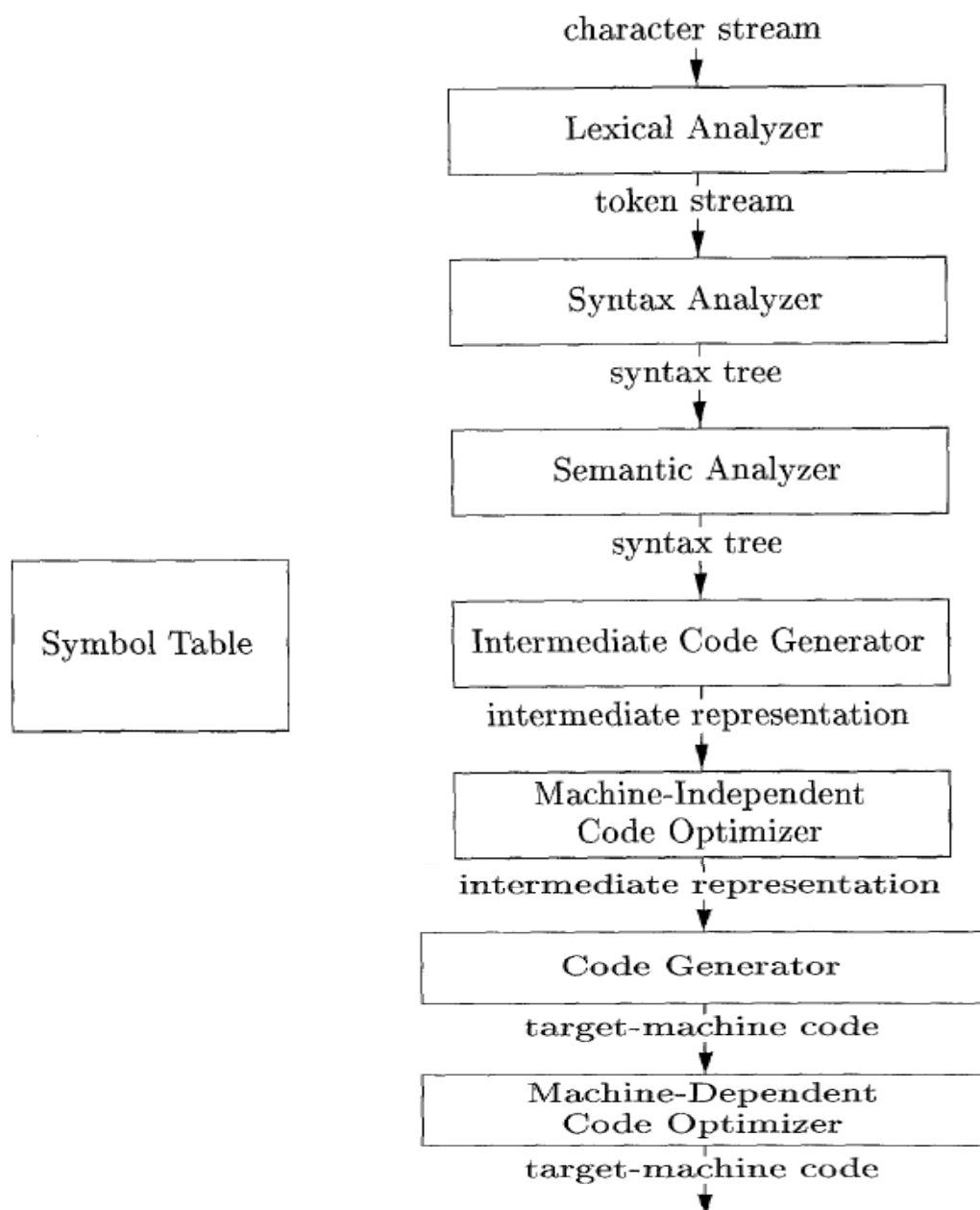


Figure.6: Phases of a compiler

## 2.1. Lexical Analysis

The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form

**(token-name, attribute-value)**

that it passes on to the subsequent phase, syntax analysis. In the token, the first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generation. For example, suppose a source program contains the assignment statement

**position = initial + rate \* 60 (1.1)**

When discussing lexical analysis, we use three related but distinct terms:

**A token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. In what follows, we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

**A pattern** is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is *matched* by many strings.

**A lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. posit ion is a lexeme that would be mapped into a token (id, 1), where id is an abstract symbol standing for identifier and 1 points to the symbol table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol = is a lexeme that is mapped into the token (=).

Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as assign for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. initial is a lexeme that is mapped into the token (id, 2), where 2 points to the symbol-table entry for initial .

4. + is a lexeme that is mapped into the token (+).

5. r a t e is a lexeme that is mapped into the token (id, 3), where 3 points to the symbol-table entry for rate .

6. \* is a lexeme that is mapped into the token (\*) .

7. 60 is a lexeme that is mapped into the token (60) .' Blanks separating the lexemes would be discarded by the lexical analyzer. Figure .7 shows the representation of the assignment statement (1.1)

After lexical analysis as the sequence of tokens

$$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle \quad (1.2)$$

In this representation, the token names =, +, and \* are abstract symbols for the assignment, addition, and multiplication operators, respectively.

### THE STRUCTURE OF A COMPILER

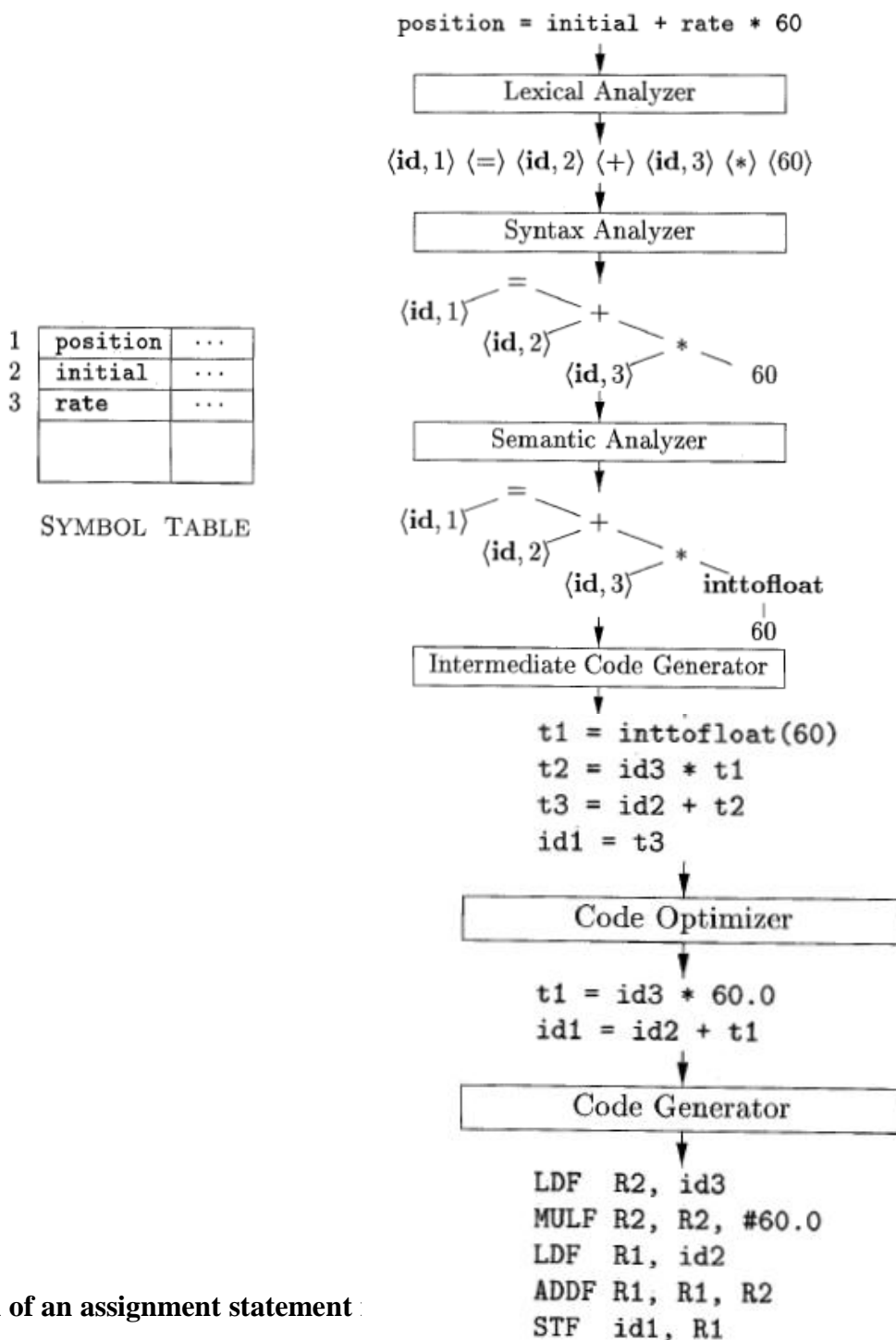


Figure.7 Translation of an assignment statement

### 2.1.1 Lexical Analysis Versus Parsing

There are a number of reasons why the analysis portion of a compiler is normally separated into lexical analysis and parsing (syntax analysis) phases.

1. Simplicity of design is the most important consideration. The separation of lexical and syntactic analysis often allows us to simplify at least one of these tasks. For example, a parser that had to deal with comments and whitespace as syntactic units would be considerably more complex than one that can assume comments and whitespace have already been removed by the lexical analyzer. If we are designing a new language, separating lexical and syntactic concerns can lead to a cleaner overall language design.
2. Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
3. Compiler portability is enhanced. Input-device-specific peculiarities can be restricted to the lexical analyzer.

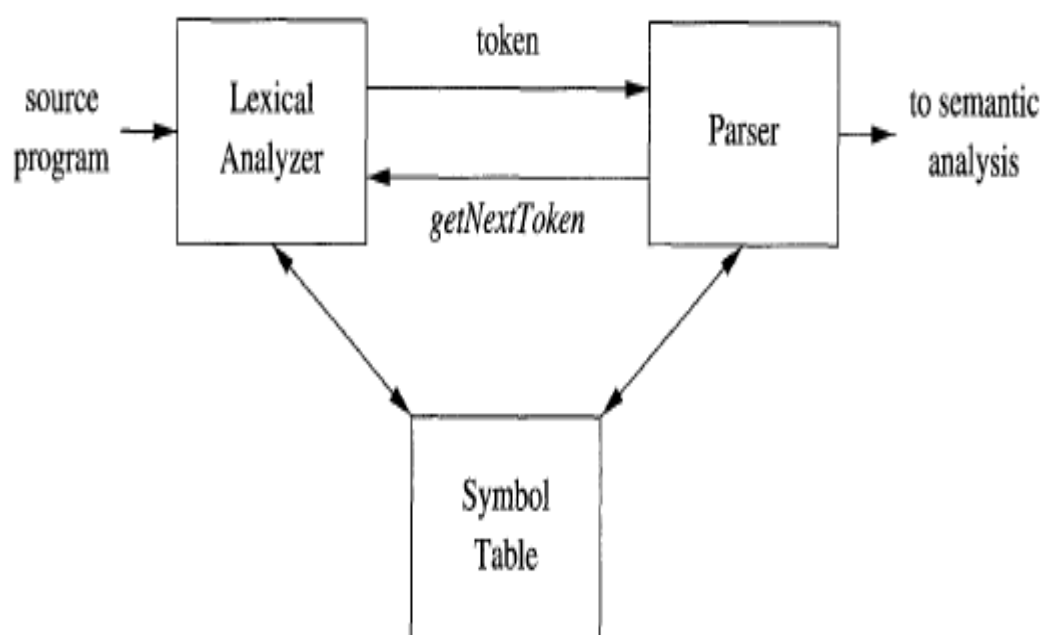


Figure.8 lexical analysis

**Example1** Figure 9 gives some typical tokens, their informally described patterns, and some sample lexemes. To see how these concepts are used in practice, in the C statement

```
printf("Total = %d\n", score);
```

both `printf` and `score` are lexemes matching the pattern for token `id`, and `"Total = %d\n"` is a lexeme matching literal.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
<b>if</b>	characters <code>i, f</code>	<code>if</code>
<b>else</b>	characters <code>e, l, s, e</code>	<code>else</code>
<b>comparison</b>	<code>&lt;</code> or <code>&gt;</code> or <code>&lt;=</code> or <code>&gt;=</code> or <code>==</code> or <code>!=</code>	<code>&lt;=</code> , <code>!=</code>
<b>id</b>	letter followed by letters and digits	<code>pi</code> , <code>score</code> , <code>D2</code>
<b>number</b>	any numeric constant	<code>3.14159</code> , <code>0</code> , <code>6.02e23</code>
<b>literal</b>	anything but <code>"</code> , surrounded by <code>"</code> 's	<code>"core dumped"</code>

Figure 9 : Examples of tokens

In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token `comparison` mentioned in Fig. 9
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal strings.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

**Example)**The following of source code is the input of Lexical Analysis phase (Fortran language using)

```
If A>=100 then
Begin
X = Y1+5.6;
Const1 =A*4;
End;
```

Token	Type	Index
if	keyword	-
A	Identifiers(id)	1
>=	Relation operators	-
100	Constant	1
then	keyword	-
Begin	keyword	-
X	Identifiers(id)	2
=	Assign operator	-
Y1	Identifiers(id)	3
+	Assign operator	-
5.6	Constant	2
;	punctuation	-
Const1	Identifiers(id)	4
=	Assign operator	-
A	Identifiers(id)	1
*	Assign operator	-
4	Constant	3
;	punctuation	-
end	keyword	-
;	punctuation	-

### Identifiers(id)

name	index
A	1
X	2
Y1	3
Const1	4

### Constant

name	index
100	1
5.6	2
4	3

### 2.1.2 Symbol Table

A symbol table is a table with two fields. A name field and an information field. This table is generally used to store information about various source language constructs. The information is collected by the analysis phase of the compiler and used by the synthesis phase to generate the target code. We required several capabilities of the symbol table we need to be able to:

1- Determine if a given name is in the table, the symbol table routines are concerned with saving and retrieving tokens.

**insert(s,t)** : this function is to add a new name to the table

**Lookup(s)** : returns index of the entry for string s, or 0 if s is not found.

2- Access the information associated with a given name, and add new information for a given name.

3- Delete a name or group of names from the t

**For example** consider tokens **begin** , we can initialize the symbol- table Using the function: **insert("begin",1)**

### 2.1.3. Attributes For Tokens

When more than one lexeme can match a pattern, the lexical analyzer must provide the subsequent compiler phases additional information about the particular lexeme that matched. For example, the pattern for token **number** matches both 0 and 1, but it is extremely important for the code generator to know which lexeme was found in the source program. Thus, in many cases the lexical analyzer returns to the parser not only a token name, but an attribute value that describes the lexeme represented by the token; the token name influences parsing decisions, while the attribute value influences translation of tokens after the parse. We shall assume that tokens have at most one associated attribute, although this attribute may have a structure that combines several pieces of information. The most important example is the

token **id**, where we need to associate with the token a great deal of information. Normally, information about an identifier - e.g., its lexeme, its type, and the location at which it is first found (in case an error message about that identifier must be issued) - is kept in the symbol table. Thus, the appropriate attribute value for an identifier is a pointer to the symbol-table entry for that identifier.

**Example 2 :** The token names 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>  
<number , integer value 2 >

Note that in certain pairs, especially operators, punctuation, and Keywords, there is no need for an attribute value. In this example, the token number has been given an integer-valued attribute. In practice, a typical compiler would instead store a character string representing the constant and use as an attribute value for number a pointer to that string.

### 2.2.1. Input Buffering

Before discussing the problem of recognizing lexemes in the input, let us examine some ways that the simple but important task of reading the source program can be speeded. This task is made difficult by the fact that we often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme.

But there are many situations where we need to look at least one additional character ahead. For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for **id**. In C, single-character operators like `-`, `=`, or `<` could also be the beginning of two-character operator like `->`, `==`, or `<=`. Thus, we shall introduce a two-buffer scheme that handles large look ahead safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

### 2.2.2.Buffer Pairs

Because of the amount of time taken to process characters and the large number of characters that must be processed during the compilation of a large source program, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

An important scheme involves two buffers that are alternately reloaded, as suggested in Fig. 10

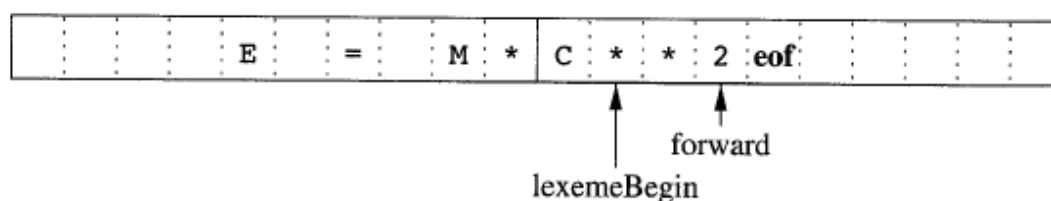


Figure 10: Using a pair of input buffers

Each buffer is of the same size  $N$ , and  $N$  is usually the size of a disk block, e.g., 4096 bytes. Using one system read command we can read  $N$  characters into a buffer, rather than using one system call per character. If fewer than  $N$  characters remain in the input file, then a special character, represented by `eof`, marks the end of the source file and is different from any possible character of the source program. Two pointers to the input are maintained:

1. Pointer lexeme Begin, marks the beginning of the current lexeme, whose extent we are attempting to determine.

2. Pointer forward scans ahead until a pattern match is found; the exact strategy whereby this determination is made will be covered in the balance of this chapter. Once the next lexeme is determined, forward is set to the character at its right end. Then, after the lexeme is recorded as an attribute value of a token returned to the parser, lexemeBegin is set to the character immediately after the lexeme just found. In Fig. 10, we see forward has passed the end of the next lexeme, \*\* (the Fortran exponentiation operator), and must be retracted one position to its left.

Advancing forward requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. As long as we never need to look so far ahead of the actual lexeme that the sum of the lexeme's length plus the distance we look ahead is greater than N, we shall never overwrite the lexeme in its buffer before determining it.

### 3. Strings and Languages

An *alphabet* is any finite set of symbols. Typical examples of symbols are letters, digits, and punctuation. The set {0,1} is the *binary alphabet*. ASCII is an important example of an alphabet; it is used in many software systems.

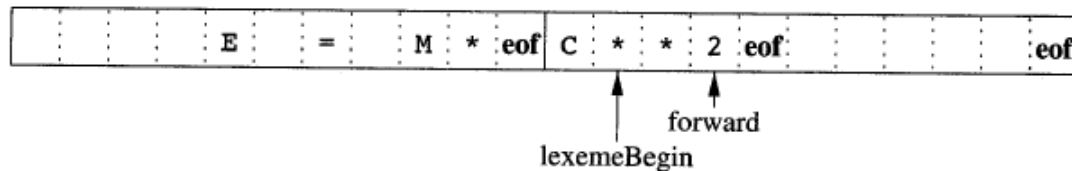


Figure 11: Sentinels at the end of each buffer

```

switch ( *forward++ ) {
case eof:
if (forward is at end of first buffer ) {
    reload second buffer;
    forward = beginning of second buffer;

else if (forward is at end of second buffer ) {
    reload first buffer;
    forward = beginning of first buffer;

else /* eof within a buffer marks the end of input */
    terminate lexical analysis;
break;
Cases for the other characters

```

Figure 12: Look ahead code with sentinels

## Languages

A language is a set of strings over some fixed alphabet. The language may contain a finite or an infinite number of strings.

Let L and M be two languages where  $L = \{\text{dog, ba, na}\}$  and  $M = \{\text{house, ba}\}$  then

- Union:  $L \cup M = \{\text{dog, ba, na, house}\}$
- Concatenation:  $LM = \{\text{doghouse, dogba, bahouse, baba, nahouse, naba}\}$
- Exponentiation:  $L^2 = LL$
- By definition:  $L^0 = \{\epsilon\}$  and  $L' = L$

The kleene closure of language L, denoted by  $L^*$ , is "zero or more Concatenation of" L.

$$L^* = L^0 \cup L' \cup L^2 \cup L^3 \dots \cup L^n \dots$$

For example, If  $L = \{a, b\}$ , then

$$L^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aba, baa, \dots\}$$

The positive closure of Language L, denoted by  $L^+$ , is "one or more Concatenation of" L.

$$L^+ = L' \cup L^2 \cup L^3 \dots \cup L^n \dots$$

For example, If  $L = \{a, b\}$ , then

$$L^+ = \{a, b, aa, ba, bb, aaa, aba, \dots\}$$

**Example:-** Let  $L$  be the set of letters  $\{A, B, \dots, Z, a, b, \dots, z\}$  and let  $D$  be the set of digits  $\{0, 1, \dots, 9\}$ . We may think of  $L$  and  $D$  in two, essentially equivalent, ways. One way is that  $L$  and  $D$  are, respectively, the alphabets of uppercase and lowercase letters and of digits. The second way is that  $L$  and  $D$  are languages, all of whose strings happen to be of length one.

1.  $L \cup D$  is the set of letters and digits - strictly speaking the language with 62 strings of length one, each of which strings is either one letter or one digit.
2.  $LD$  is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3.  $L^4$  is the set of all 4-letter strings.
4.  $L^*$  is the set of all strings of letters, including  $\epsilon$ , the empty string.
5.  $L(L \cup D)^*$  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.

### 3.1.1.Regular Definitions

For notational convenience, we may wish to give names to certain regular expressions and use those names in subsequent expressions, as if the names were themselves symbols. If  $C$  is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

$$\begin{array}{lll} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ & \dots & \\ d_n & \rightarrow & r_n \end{array}$$

**where:**

1. Each  $d_i$  is a new symbol, not in  $C$  and not the same as any other of the  $d$ 's, and
2. Each  $r_i$  is a regular expression over the alphabet  $C \cup \{d_1, d_2, \dots, d_{i-1}\}$ .

**Example :**

**keyword** = BEGIN | END | IF | THEN | ELSE

**Identifier** = letter (letter | digit)\*

**constant** = digit+

**relop** = < | <= | = | <> | > | >=

**Q1) Design the regular expression (r.e.) for the language accepting all combinations of a's except the null string over = { a }**

$$L = \{a, aa, aaa, \dots\}$$

$$R = a^+$$

**Q2) Construct the r.e for the language accepting all the strings which are ending with 00 over the set = { 0, 1 }.**

$$\text{r.e.} = (0+1)^* 00$$

## Identity Rules

$$1- \epsilon R = R \epsilon = R$$

$$2- \epsilon^* = \epsilon \quad \epsilon \text{ is null string}$$

$$3- (\emptyset)^* = \epsilon$$

$$4- \emptyset R = R\emptyset = \emptyset$$

$$5- \emptyset + R = R$$

$$6- R + R = R$$

$$7- RR^* = R^*R = R^+$$

$$8- (R^*)^* = R^*R = R^+$$

$$9- \epsilon + RR^* = R^*$$

$$10- (P+Q)R = PR + QR$$

$$11- (P+Q)^* = (P^*Q^*) = (P^* + Q^*)^*$$

$$12- R^* (\epsilon + R) = (\epsilon + R) R^* = R^*$$

$$13- (R + \epsilon)^* = R^*$$

$$14- \epsilon + R^* = R^*$$

$$15- (PQ)^* P = P(QP)^*$$

$$16- R^*R + R = R^* R$$

EXPRESSION	MATCHES	EXAMPLE
$c$	the one non-operator character $c$	a
$\backslash c$	character $c$ literally	$\backslash *$
$"s"$	string $s$ literally	$"**"$
$.$	any character but newline	a.*b
$^$	beginning of a line	$^abc$
$\$$	end of a line	abc\$
$[s]$	any one of the characters in string $s$	[abc]
$[^s]$	any one character not in string $s$	[^abc]
$r^*$	zero or more strings matching $r$	a*
$r^+$	one or more strings matching $r$	a+
$r^?$	zero or one $r$	a?
$r\{m,n\}$	between $m$ and $n$ occurrences of $r$	a[1,5]
$r_1r_2$	an $r_1$ followed by an $r_2$	ab
$r_1 \mid r_2$	an $r_1$ or an $r_2$	a b
$(r)$	same as $r$	(a b)
$r_1/r_2$	$r_1$ when followed by $r_2$	abc/123

### 3.1.2. Arden's Theorem

Let, P and Q be the two regular expressions over the input set  $\Sigma$ . The regular expression R is given as

$$R = Q + RP$$

Which has a unique solution as  $R = QP^*$ .

**Proof :** Let, P and Q are two regular expressions over the input string  $\Sigma$ .

If P does not contain  $\Sigma$  then there exists R such that.

$$R = Q + RP$$

We will replace R by  $QP^*$  in equation 1.

Consider R.H.S of equation 1.

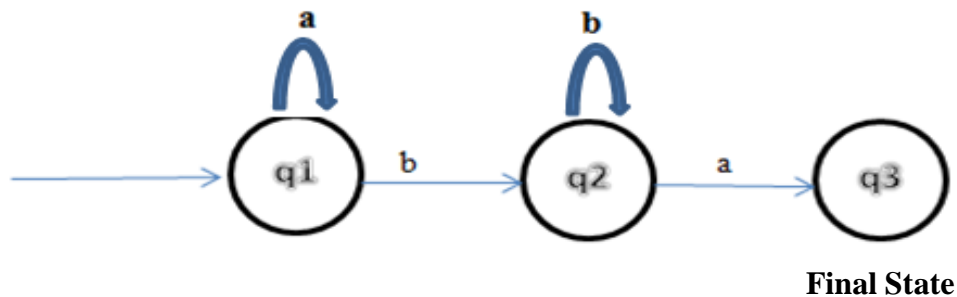
$$= Q + QP^* P$$

$$= Q (\epsilon + P^* P)$$

$$= QP^*$$

$$\text{Thus } R = QP^*$$

EX1 :- Construct Regular Expression from given DFA(Using Arden's Theorem)



**Sol:-**

$$q1 = \epsilon + q1a$$

State= input coming to it\* source state of input

**Similarly**

$$q2 = q1b + q2b$$

Let us simplify q1 first

$$q1 = \epsilon + q1a$$

We can re-write it as

$$q1 = \epsilon + q1a$$

Which similar to  $R = Q + RP$  which further gets reduced to  $R = QP^*$  Assuming

$$R = q1, Q = \epsilon, P = a$$

**We Get**

$$q1 = \epsilon.a^*$$

$$q1 = a^* \quad : \epsilon.R = R$$

Substituting value of q1 in q2 we get

$$q2 = q1b + q2b$$

$$q2 = a^*b + q2b$$

We can compare this equation with  $R = Q + RP$  Assuming  $R = q2, Q = a^*b, P = b$  which gets reduced to  $R = QP^*$

$$q2 = a^*b . b^* \quad \text{As } R^*R = R +$$

$$q2 = a^* . b +$$