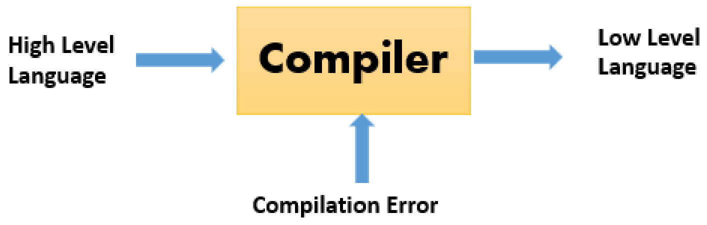
COMPILER DESIGN CHAPTER – 1

1. INTRODUCTION

A compiler is a computer program which helps you transform source code written in a high-level language into low-level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code. The compiler also makes the end code efficient, which is optimized for execution time and memory space.

The compiling process includes basic translation mechanisms and error detection. The compiler process goes through lexical, syntax, and semantic analysis at the front end and code generation and optimization at the back-end. The compiler may be of single pass, two pass and multi pass compilers.



The main tasks performed by the Compiler are:

- Breaks up the source program into pieces and impose grammatical structure on them.
- Allows you to construct the desired target program from the intermediate representation and also create the symbol table.
- Compiles source code and detects errors in it.
- Manage storage of all variables and codes.
- Support for separate compilation
- Read, analyze the entire program, and translate to semantically equivalent.
- Translating the source code into object code depending upon the type of machine

Features of Compiler:

- Correctness
- Speed of compilation
- Preserve the correct the meaning of the code
- The speed of the target code
- Recognize legal and illegal program constructs.

- Good error reporting/handling
- Code debugging help.

2. ANALYSIS OF SOURCE PROGRAM

In compiling, analysis consists of three phases:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis

- **Lexical Analysis:** Lexical analysis is the starting phase of the compiler. It gathers modified source code that is written in the form of sentences from the language preprocessor. The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by removing whitespace in the source code. If the lexical analyzer gets any invalid token, it generates an error. The stream of character is read by it and it seeks the legal tokens, and then the data is passed to the syntax analyzer.

  In a compiler linear analysis is called lexical analysis or scanning. The lexical analysis phase reads the characters in the source program and grouped into them tokens that are sequence of characters having a collective meaning.

Example :

position : = initial + rate * 60

Identifiers – position, initial, rate.

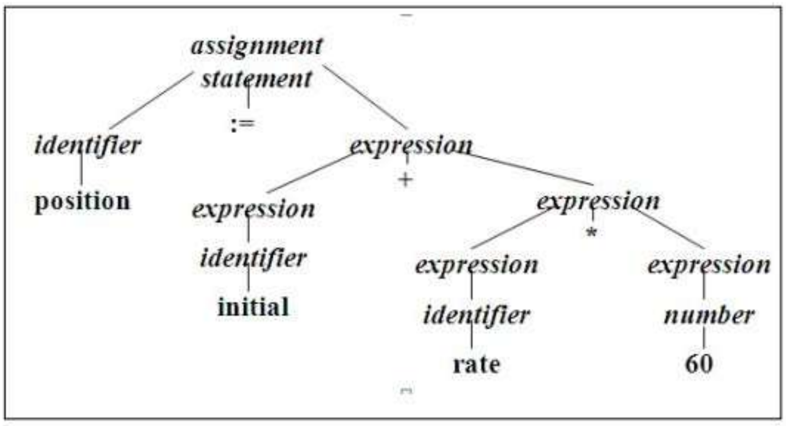
Operators - + , *

Assignment symbol - : =

Number – 60

- **Syntax Analysis:** Syntax Analysis or Hierarchical analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree, the input string is found to be in the correct syntax. if not, the error is reported by the syntax analyzer.

It involves grouping the tokens of the source program into grammatical phrases that are used by the complier to synthesize output. A **syntax tree** is the tree generated as a result of syntax analysis in which the interior nodes are the operators, and the exterior nodes are the operands.



- **Semantic Analysis:** is the third phase of Compiler. Semantic Analysis makes sure that declarations and statements of program are semantically correct. It is a collection of procedures which is called by parser as and when required by grammar. Both syntax tree of previous phase and symbol table are used to check the consistency of the given code.

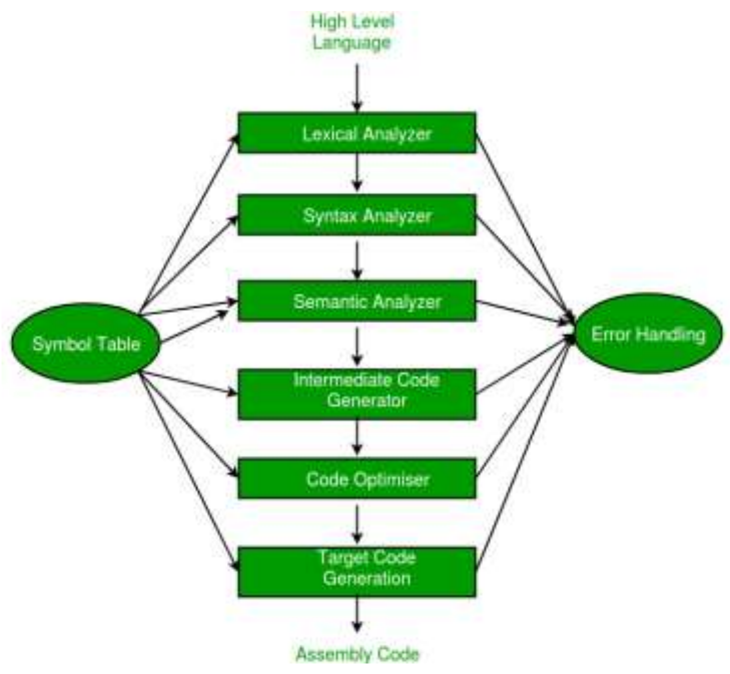
  This phase checks the source program for semantic errors and gathers type information for subsequent code generation phase. An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification.

  Errors recognized by semantic analyzer are as follows:
  - Type mismatch
  - Undeclared variables
  - Reserved identifier misuse

3. PHASES OF COMPILER

Due to the complexity of compilation task, a Compiler typically proceeds in a Sequence of compilation phases. The phases communicate with each other via clearly defined interfaces.

Generally, an interface contains a Data structure. Each phase works on an abstract intermediate representation of the source program, not the source program text itself. Compiler Phases are the individual modules which are chronologically executed to perform their respective Sub-activities, and finally integrate the solutions to give target code. It is desirable to have relatively few phases, since it takes time to read and write immediate files. A Compiler is having the following Phases:



In addition to these, it also has Symbol table management, and Error handler phases. Not all the phases are mandatory in every Compiler. e.g, Code Optimizer phase is optional in some cases. The Phases of compiler divided in to two parts, first three phases we are called as Analysis part remaining three called as Synthesis part.

We basically have two phases of compilers, namely the Analysis phase and Synthesis phase. The analysis phase creates an intermediate representation from the given source code. The synthesis phase creates an equivalent target program from the intermediate representation.

- **Symbol Table –** It is a data structure being used and maintained by the compiler, consisting of all the identifier's names along with their types. It helps the compiler to function smoothly by finding the identifiers quickly.
- **LEXICAL ANALYZER (SCANNER):** The Scanner is the first phase that works as interface between the compiler and the Source language program and performs the following functions:

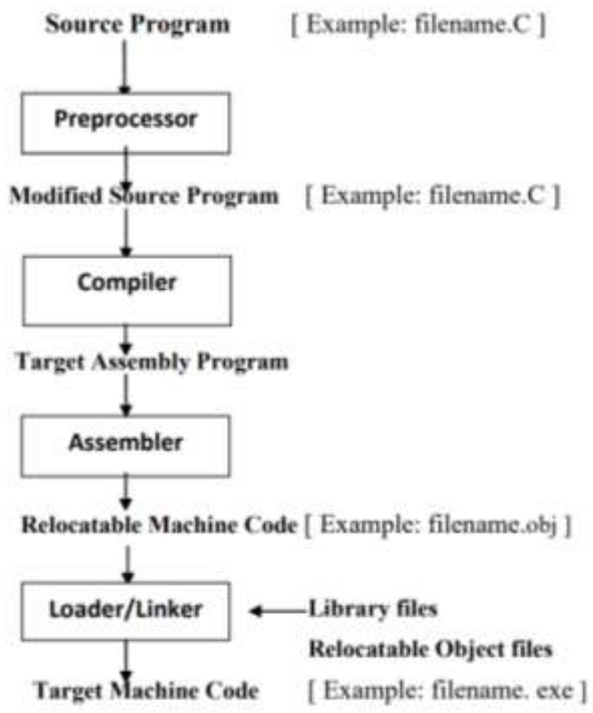
  - Reads the characters in the Source program and groups them into a stream of tokens in which each token specifies a logically cohesive sequence of characters, such as an identifier, a Keyword, a punctuation mark, a multi character operator like := .
  - The character sequence forming a token is called a lexeme of the token.
  - The Scanner generates a token-id, and also enters that identifiers name in the Symbol table if it doesn't exist.
  - Also removes the Comments, and unnecessary spaces. The format of the token is < Token name, Attribute value>
- **SYNTAX ANALYZER (PARSER):** The Parser interacts with the Scanner, and its subsequent phase Semantic Analyzer and performs the following functions:
  - Groups the above received, and recorded token stream into syntactic structures, usually into a structure called Parse Tree whose leaves are tokens.
  - The interior node of this tree represents the stream of tokens that logically belong together.
  - It means it checks the syntax of program elements.
- **SEMANTIC ANALYZER**: This phase receives the syntax tree as input and checks the semantically correctness of the program. Though the tokens are valid and syntactically correct, it may happen that they are not correct semantically. Therefore, the semantic analyzer checks the semantics (meaning) of the statements formed.
  - The Syntactically and Semantically correct structures are produced here in the form of a Syntax tree or some other sequential representation like matrix.
- **INTERMEDIATE CODE GENERATOR(ICG):** This phase takes the syntactically and semantically correct structure as input and produces its equivalent intermediate notation of the source program. The Intermediate Code should have two important properties specified below:
  - It should be easy to produce, and Easy to translate into the target program. Example intermediate code forms are:

    - Three address codes,
    - Polish notations, etc.
- **CODE OPTIMIZER**: This phase is optional in some Compilers, but so useful and beneficial in terms of saving development time, effort, and cost. This phase performs the following specific functions:
    - Attempts to improve the IC so as to have a faster machine code. Typical functions include –Loop Optimization, Removal of redundant computations, Strength reduction, Frequency reductions etc.
    - Sometimes the data structures used in representing the intermediate forms may also be changed.
- **CODE GENERATOR**: This is the final phase of the compiler and generates the target code, normally consisting of the relocatable machine code or Assembly code or absolute machine code.
    - Memory locations are selected for each variable used, and assignment of variables to registers is done.
    - Intermediate instructions are translated into a sequence of machine instructions.

The Compiler also performs the Symbol table management and Error handling throughout the compilation process. Symbol table is nothing but a data structure that stores different source language constructs, and tokens generated during the compilation. These two interact with all phases of the Compiler.

4. COUSINS OF COMPILER (LANGUAGE PROCESSING SYSTEM)

- **Preprocessor**: The preprocessor is considered as a part of the Compiler. It is a tool which produces input for Compiler. It deals with macro processing, augmentation, language extension, etc.
- **Interpreter**: An interpreter is like a Compiler which translates high-level language into low-level machine language. The main difference between both is that the interpreter reads and transforms code line by line. Compiler reads the entire code at once and creates the machine code.



- **Assembler**: It translates assembly language code into machine understandable language. The output result of assembler is known as an object file which is a combination of machine instruction as well as the data required to store these instructions in memory.
- **Linker**: The linker helps you to link and merge various object files to create an executable file. All these files might have been compiled with separate assemblers. The main task of a linker is to search for called modules in a program and to find out the memory location where all modules are stored.
- **Loader**: The loader is a part of the OS, which performs the tasks of loading executable files into memory and run them. It also calculates the size of a program which creates additional memory space.
- **Cross-compiler**: A Cross compiler in compiler design is a platform which helps you to generate executable code.
- **Source-to-source Compiler**: Source to source compiler is a term used when the source code of one programming language is translated into the source of another language.

5. GROUPING OF PHASES

Several phases are grouped together to a pass so that it can read the input file and write an output file.

1. One-Pass – In One-pass all the phases are grouped into one phase. The six phases are included here in one pass.
2. Two-Pass – In Two-pass the phases are divided into two parts i.e. Analysis or Front End part of the compiler and the synthesis part or back end part of the compiler.

6. COMPILER CONSTRUCTION TOOLS

Compiler construction tools were introduced as computer-related technologies spread all over the world. They are also known as compilers- compilers, compiler- generators or translator.

These tools use specific language or algorithm for specifying and implementing the component of the compiler. The following are the examples of compiler construction tools.

- **Scanner generators**: This tool takes regular expressions as input. For example, LEX for Unix Operating System.
- **Syntax-directed translation engines**: These software tools offer an intermediate code by using the parse tree. It has a goal of associating one or more translations with each node of the parse tree.
- **Parser generators:** A parser generator takes a grammar as input and automatically generates source code which can parse streams of characters with the help of a grammar.
- **Automatic code generators**: Takes intermediate code and converts them into Machine Language.
- **Data-flow engines**: This tool is helpful for code optimization. Here, information is supplied by the user, and intermediate code is compared to analyze any relation. It is also known as data-flow analysis. It helps you to find out how values are transmitted from one part of the program to another part.

7. LEXICAL ANALYSIS

Lexical analysis is the starting phase of the compiler, also known as a scanner. It converts the High-level input program into a sequence of Tokens. It gathers modified source code that is written in the form of sentences from the language preprocessor. The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by removing whitespace and comments in the source code. If the lexical analyzer gets any invalid token, it generates an error. The stream of characters is read by it and it seeks the legal tokens, and then the output is a sequence of characters passed to the parser for syntax analysis.

Programs that perform Lexical Analysis in compiler design are called lexical analyzers or lexers. A lexer contains a tokenizer or scanner.

Advantages of Lexical Analysis

- Lexical analysis helps the browsers to format and display a web page with the help of parsed data from JavsScript, HTML, CSS.
- It is responsible for creating a compiled binary executable code.
- It helps to create a more efficient and specialized processor for the task.

Disadvantages of Lexical Analysis

- It requires additional runtime overhead to generate the lexer table and construct the tokens.
- It requires much effort to debug and develop the lexer and its token description.
- Significant time is required to read the source code and partition it into tokens.

i. LEXEME

A sequence of characters in the source code, as per the matching pattern of a token, is known as lexeme. It is also called the instance of a token.

ii. PATTERNS

The description used by the token is known as a pattern. In the case of a keyword which is used as a token, the pattern is a sequence of characters.

iii. TOKENS

It is a sequence of characters that represents a unit of information in the source code. A lexeme must follow certain predefined rules to be considered a valid token. These rules are defined using a pattern. These patterns represent the token, and regular expressions define the patterns. Keywords, numbers, punctuation, strings, and identifiers can be considered tokens in any programming language. Comments, preprocessor directive, macros, blanks, tabs, newline, etc. are considered as non-tokens.

Consider a line as

```
#include <stdio.h>
int maximum(int x, int y) {
   // This will compare two numbers
   if (y > x)
      return y;
   else {
      return x;
   }
}
```

Then the lexemes tokens are as below:

:

Lexeme	**Token**
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword

Y	Identifier
)	Operator
{	Operator
If	Keyword

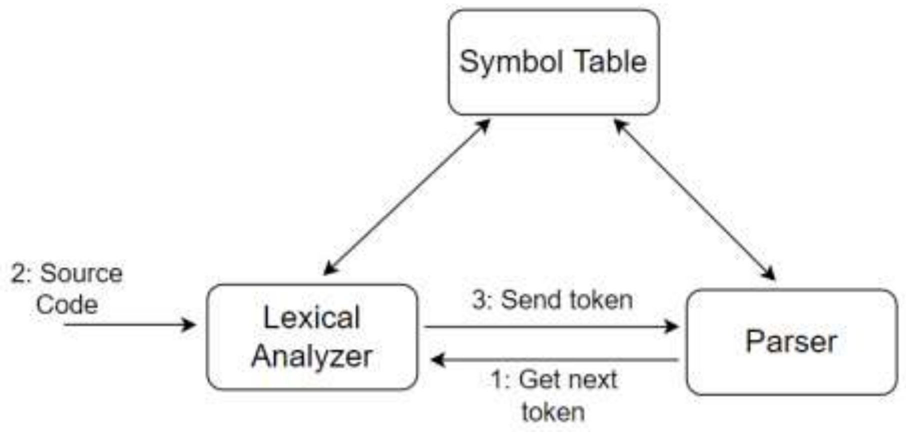
The corresponding non-token table is shown below:

Type	Examples
Comment	//Example
Pre-processor directive	#include<stdio.h>

iv. LEXICAL ANALYZER

To read the input character in the source code and produce a token is the most important task of a lexical analyzer. The lexical analyzer goes through the entire source code and identifies each token one by one. The scanner is responsible for producing tokens when it is requested by the parser. The lexical analyzer avoids the whitespace and comments while creating these tokens. If any error occurs, the analyzer correlates these errors with the source file and line number.

The lexical analyzer identifies the error with the help of the automation machine and the grammar of the given language on which it is based like C, C++, and gives row number and column number of the error.



How Lexical Analyzer works-

- **Input preprocessing**: This stage involves cleaning up the input text and preparing it for lexical analysis. This may include removing comments, whitespace, and other non-essential characters from the input text.
- **Tokenization:** This is the process of breaking the input text into a sequence of tokens. This is usually done by matching the characters in the input text against a set of patterns or regular expressions that define the different types of tokens.
- **Token classification**: In this stage, the lexer determines the type of each token. For example, in a programming language, the lexer might classify keywords, identifiers, operators, and punctuation symbols as separate token types.
- **Token validation**: In this stage, the lexer checks that each token is valid according to the rules of the programming language. For example, it might check that a variable name is a valid identifier, or that an operator has the correct syntax.
- **Output generation**: In this final stage, the lexer generates the output of the lexical analysis process, which is typically a list of tokens. This list of tokens can then be passed to the next stage of compilation or interpretation.

Exercise 1: Count number of tokens :

```
int main()
{
  int a = 10, b = 20;
  printf("sum is :%d",a+b);
  return 0;
}
```

Answer: Total number of tokens: 27.

Exercise 2: Count number of tokens: int max(int i);

Lexical analyzer first read int and finds it to be valid and accepts as token max is read by it and found to be a valid function name after reading (int is also a token , then again i as another token and finally ;

Answer: Total number of tokens 7:

int, max, (,int, i,), ;

v. ROLE OF LEXICAL ANALYZER

Lexical analyzer performs below given tasks:

- Lexical analyzers are a bridge between the source code and the other compiler phases.
- While sending tokens to the parser, the lexical analyzer communicates with the symbol table.
- Helps to identify tokens into the symbol table.
- Removes white spaces and comments from the source program.
- Correlates error messages with the source program
- Helps you to expand the macros if it is found in the source program.
- Read input characters from the source program.

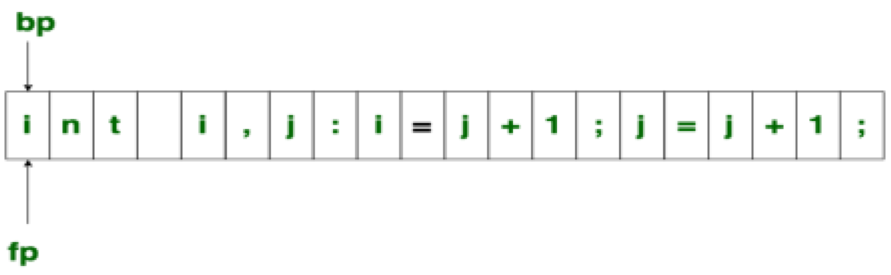
8. INPUT BUFFERING

Lexical Analysis has to access secondary memory each time to identify tokens. It is time-consuming and costly. So, the input strings are stored into a buffer and then scanned by Lexical Analysis.

Lexical Analysis scans input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens −

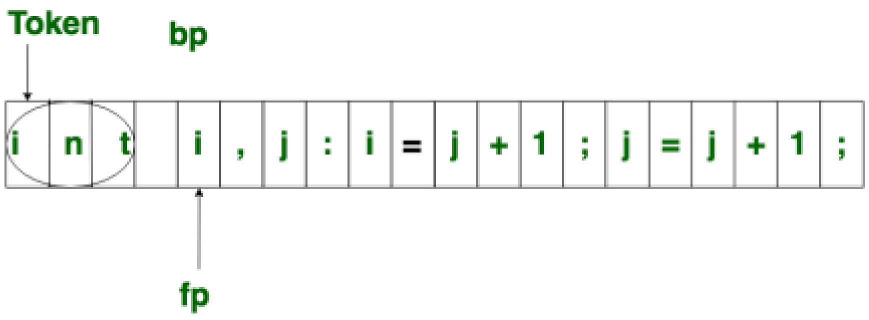
- **Begin Pointer (bptr)** − It points to the beginning of the string to be read.
- **Look Ahead Pointer (lptr)** − It moves ahead to search for the end of the token.

Initially both the pointers point to the first character of the input string as shown below



Initial Configuration

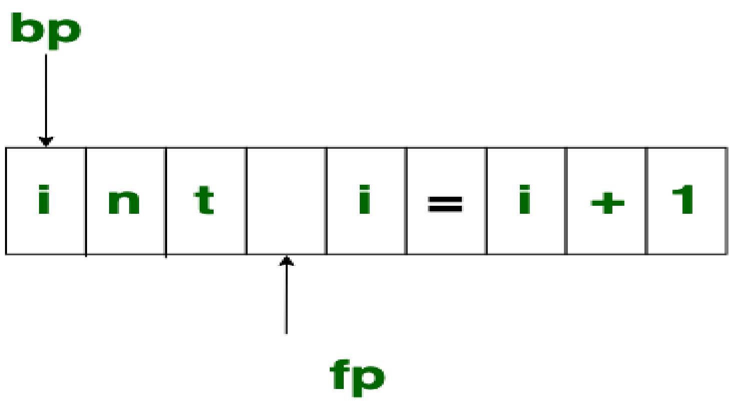
The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. In the above example as soon as ptr (fp) encounters a blank space the lexeme “int” is identified. The fp will be moved ahead at white space, when fp encounters white space, it ignores and moves ahead. then both the begin ptr(bp) and forward ptr(fp) are set at next token.



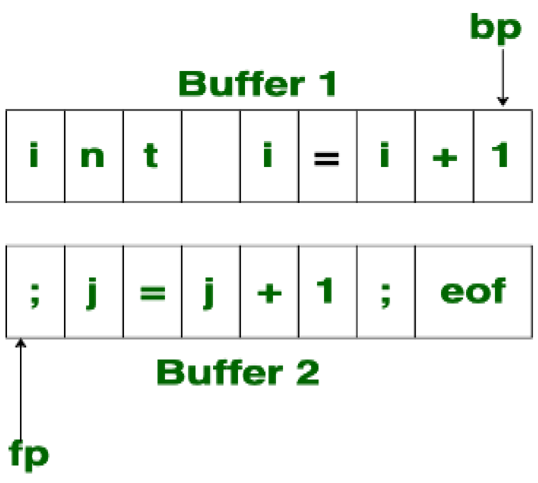
Input buffering

The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. Hence buffering technique is used. A block of data is first read into a buffer, and then second by lexical analyzer. There are two methods used in this context: One Buffer Scheme, and Two Buffer Scheme.

One Buffer Scheme: In this scheme, only one buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.



Two Buffer Scheme: To overcome the problem of one buffer scheme, in this method two buffers are used to store the input string. the first buffer and second buffer are scanned alternately. when end of current buffer is reached the other buffer is filled. the only problem with this method is that if length of the lexeme is longer than length of the buffer then scanning input cannot be scanned completely. Initially both the bp and fp are pointing to the first character of first buffer. Then the fp moves towards right in search of end of lexeme. as soon as blank character is recognized, the string between bp and fp is identified as corresponding token. To identify, the boundary of first buffer end of buffer character should be placed at the end first buffer. Similarly end of second buffer is also recognized by the end of buffer mark present at the end of second buffer. When fp encounters first *eof*, then one can recognize end of first buffer and hence filling up second buffer is started. in the same way when second *eof* is obtained then it indicates of second buffer. Alternatively both the buffers can be filled up until end of the input program and stream of tokens is identified. This eof character introduced at the end is calling *Sentinel* which is used to identify the end of buffer.



9. SPECIFICATION OF TOKENS

Specification of tokens depends on the pattern of the lexeme. Here we will be using regular expressions to specify the different types of patterns that can actually form tokens.

- String and Languages
- Operation on Languages
- Regular Expression
- Regular Definition

i. String and Languages

***String**:* The string is a finite set of alphabets. Alphabet is a finite set of symbols. Symbols can be letters, digits and punctuation.

If you can remember ASCII system that is used in almost every computer, denotes the alphabet A using the set of digits {0, 1} i.e. A = 01000001.

Language: Language is a set of strings over some fixed alphabets. Like the English language is a set of strings over the fixed alphabets 'a to z'.

ii. Operation on Languages

As we have learnt language is a set of strings that are constructed over some fixed alphabets. Now the operation that can be performed on languages are:

- **Union**

Union is the most common set operation. Consider the two languages L and M. Then the union of these two languages is denoted by:

L [∪ M = { s | s is in L or s is in M}

That means the string s from the union of two languages can either be from language L or from language M.

If L = {a, b} and M = {c, d}Then L ∪ M = {a, b, c, d}

- **Concatenation**

Concatenation links the string from one language to the string of another language in a series in all possible ways. The concatenation of two different languages is denoted by:

L · M = {st | s is in L and t is in M}If L = {a, b} and M = {c, d}

Then L · M = {ac, ad, bc, bd}

- **Kleene Closure**

Kleene closure of a language L provides you with a set of strings. This set of strings is obtained by concatenating L zero or more time. The Kleene closure of the language L is denoted by:

If L = {a, b}L* = {∈, a, b, aa, bb, aaa, bbb, …}

- **Positive Closure**

The positive closure on a language L provides a set of strings. This set of strings is obtained by concatenating ‘L’ one or more times. It is denoted by:

It is similar to the Kleene closure. Except for the term L0, i.e. L+ excludes ∈ until it is in L itself.

If L = {a, b}L+ = {a, b, aa, bb, aaa, bbb, …}

So, these are the four operations that can be performed on the languages in the lexical analysis phase.

iii. Regular Expression

A regular expression is a sequence of symbols used to specify lexeme patterns. A regular expression is helpful in describing the languages that can be built using operators such as union, concatenation, and closure over the symbols.

iv. Regular Definition

The regular definition is the name given to the regular expression. The regular definition (name) of a regular expression is used in the subsequent expressions. The regular definition used in an expression appears as if it is a symbol.

If Σ = alphabet of the basic symbol

Then regular definition is a sequence of definitions of the form.

d1 -> r1d2 -> r2

…

dn -> rn

10. DATA STRUCTURES IN COMPILATION

During compilation, the symbol table is searched each time an identifier is encountered. Data are added if a new name or new information about an existing name is find. Thus, in designing a symbol table structure, it would like a scheme that enables us to insert new entries and identify current entries in a table effectively.

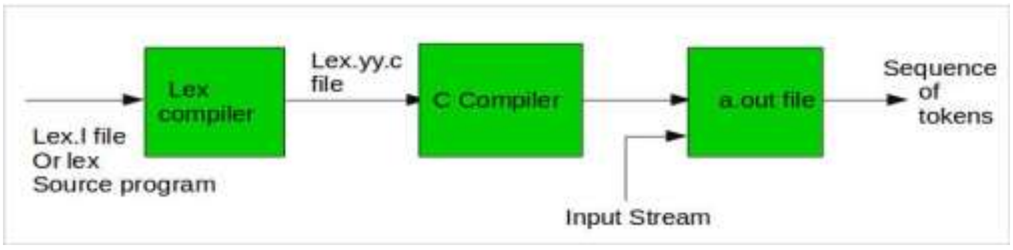
There are four symbol tables used in a data structure which are as follows −

- **Lists−** The simplest and clear to implement a data structure for a symbol is the linear list of records.
  - In this method, an array is used to store names and associated information.
  - A pointer "available" is maintained at end of all stored records and new names are added in the order as they arrive
  - To search for a name we start from the beginning of the list till available pointer and if not found we get an error "use of the undeclared name"
  - While inserting a new name we must ensure that it is not already present otherwise an error occurs i.e. "Multiple defined names"
  - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
  - The advantage is that it takes a minimum amount of space.
- **Linked List –**
  - This implementation is using a linked list. A link field is added to each record.
  - Searching of names is done in order pointed by the link of the link field.
  - A pointer "First" is maintained to point to the first record of the symbol table.
  - Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
- **Hash Table –**
  - In hashing scheme, two tables are maintained – a hash table and symbol table and are the most commonly used method to implement symbol tables.
  - A hash table is an array with an index range: 0 to table size – 1. These entries are pointers pointing to the names of the symbol table.
  - To search for a name we use a hash function that will result in an integer between 0 to table size – 1.
  - Insertion and lookup can be made very fast – O(1).
  - The advantage is quick to search is possible and the disadvantage is that hashing is complicated to implement.

- **Binary Search Tree –**
  - Another approach to implementing a symbol table is to use a binary search tree i.e. we add two link fields i.e. left and right child.
  - All names are created as child of the root node that always follows the property of the binary search tree.
  - Insertion and lookup are O(log2 n) on average.

11. LEX – LEXICAL ANALYZER GENERATOR

LEX is a tool that generates a lexical analyzer program for a given input string. It processes the given input string/file and transforms them into tokens. It is used by YACC programs to generate complete parsers.



- An input file, which we call *l e x . l* , is written in the Lex language and describes the lexical analyzer to be generated.
- The Lex compiler transforms *l e x.l* to a C program, in a file that is always named *l e x . y y . c*.
- The latter file is compiled by the C compiler into a file called *a . o u t.*
- The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

Parts of LEX Program:

The layout of a LEX source program is:

- Definitions
- Rules
- Auxiliary routines

A double modulus sign separates each section %% --- %%.

Definitions:

The definitions are at the top of the LEX source file. It includes the regular definitions, the C, directives, and any global variable declarations. The code specific to C is placed within %{ and %}

This are instructions for the C compiler. They are used for include header files, defining global variables and constants and declaration of functions. They consist of two parts, auxiliary declarations and regular definitions. They are not processed by the lex tool instead are copied by the lex to the output file *lex.yy.c* file.

An example of auxiliary declarations

```
%{
   #include<stdio.h>
   int global_variable;
%}
```

An example of Regular definitions

```
number [0-9]+
op[-|+|*|/|^|=]
```

Rules:

This section may contain multiple rules. Each rule consists of:

- A regular expression (name)
- A piece of code (output action)

They execute whenever a token in the input stream matches with the grammar in the form: pattern action and pattern must be unintended and action begin on the same line in {} brackets. The rule section is enclosed in “%%   %%”.

For example

```
%%
   {number} {printf("" number");}
   {op} {printf(" operator");}
%%
```

Pattern	It can match with
[0-9]	all the digits between 0 and 9
[0+9]	either 0, + or 9

Pattern	It can match with
[0, 9]	either 0, ‘, ‘ or 9
[0 9]	either 0, ‘ ‘ or 9
[-09]	either -, 0 or 9
[-0-9]	either – or all digit between 0 and 9
[0-9]+	one or more digit between 0 and 9
[^a]	all the other characters except a
[^A-Z]	all the other characters except the upper-case letters
a{2, 4}	either aa, aaa or aaaa
a{2, }	two or more occurrences of a
a{4}	exactly 4 a’s i.e, aaaa
.	any character except newline
a*	0 or more occurrences of a
a+	1 or more occurrences of a
[a-z]	all lower case letters
[a-zA-Z]	any alphabetic letter
w(x | y)z	wxz or wyz

Auxiliary routines:

This section includes functions that may be required in the rules section. Here, a function is written in regular C syntax. Most simple lexical analyzers only require the main () function.

The *yytext* keyword gives the current lexeme. The generated code is placed into a function called *yylex().* The main() function always calls the *yylex()* function. The programmer can also implement additional functions used for actions. These functions are compiled separately and loaded with lexical analyzer.

For example,

```
/* declarations */
%%
/* rules */
%%
/* functions */
int main()
{
   yylex();
   return 1;
}
```

yy Variables:

These are variables given by the lex which enable the programmer to design a sophisticated lexical analyzer. They include yyin which points to the input file, it points to the input file set by the programmer, if not assigned, it defaults to point to the console input(stdin), yytext which will hold the lexeme currently found and yyleng which is a int variable that stores the length of the lexeme pointed to by yytext.

Each invocation of yylex() function will result in a yytext which carries a pointer to the lexeme found in the input stream yylex(). This is overwritten on each yylex() function invocation.

yylex() is defined by lex in lex.yy.c but it not called by it. It is called in the auxilliary functions section in the lex program and returns an int. Code generated by the lex is defined by yylex() function according to the specified rules.

yywrap() is defined in the auxilliary function section. It is called by the yylex() function when end of input is encountered and has an int return type. If the function returns a non-zero(true), yylex() will terminate the scanning process and returns 0, otherwise if yywrap() returns 0(false), yylex() will assume that there is more input and will continue scanning from location pointed at by yyin.

% option noyywrap is declared in the declarations section to avoid calling of yywrap() in lex.yy.c file. It is mandatory to either define yywrap() or indicate its absence using the describe option above.

How to run the Program

To run the program, it should be first saved with the extension .l or .lex. Run the below commands on terminal in order to run the program file.

Step 1: lex filename.l or lex filename.lex depending on the extension file is saved with

Step 2: gcc lex.yy.c

Step 3: a.exe for Windows and ./a.out for Linux

Step 4: Provide the input to the program in case it is required.

Example 1: Count the number of characters in a string

```
/*** Definition Section has one variable which can be accessed inside
yylex() and main() ***/

        %{
        int count = 0;
        %}

/*** Rule Section has three rules, first rule matches with capital
letters, second rule matches with any character except newline and third
rule does not take input after the enter***/
        %%
        [A-Z] {printf("%s capital letter\n", yytext);
               count++;}
        .     {printf("%s not a capital letter\n", yytext);}
        \n    {return 0;}
        %%
```

```
/*** Code Section prints the number of capital letter present in the
given input***/
       int yywrap(){}
       int main(){

// Explanation:
// yywrap() - wraps the above rule section
/* yyin - takes the file pointer which contains the input*/
/* yylex() - this is the main flex function which runs the Rule Section*/
// yytext is the text in the buffer

       // Uncomment the lines below to take input from file
       // FILE *fp;
       // char filename[50];
       // printf("Enter the filename: \n");
       // scanf("%s",filename);
       // fp = fopen(filename,"r");
       // yyin = fp;

yylex();
printf("\nNumber of Capital letters in the given input - %d\n", count);

return 0;
}
```

Example 2 : The given code is a LEX program that converts every decimal number in a given input to hexadecimal.

```
%{
#include <stdlib.h>
#include <stdio.h>
int count = 0;
%}

%%
[0-9]+ { int no = atoi(yytext);
         printf("%x",no);
         count++;
      }
[\n]    return 0;
%%

int main(void)
```

```
{
        printf("Enter any number(s) to be converted to hexadecimal:\n");
        yylex();
        printf("\n");
        return 0;
}
```