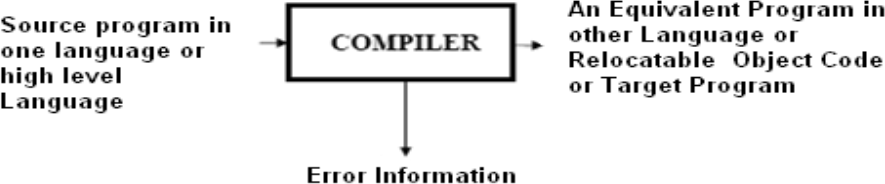
1. **What is a Compiler? Explain the working of a Compiler with your own example?**

**A. Compiler :** Compiler is a program, reads program in one language called Source Language and translates in to its equivalent program in another Language called Target Language, in addition to this its presents the error information to the User.



If the target program is an executable machine-language program, it can then be called by the users to process inputs and produce outputs.

**Target Program**

**Input Output**

**2. Explain the Language Processing System.**

A. Based on the input the translator takes and the output it produces, a language translator can be called as any one of the following.

**Preprocessor:** A preprocessor takes the skeletal source program as input and produces an extended version of it, which is the resultant of expanding the Macros, manifest constants if any, and including header files etc in the source file. For example, the C preprocessor is a macro processor that is used automatically by the C compiler to transform our source before actual compilation. Over and above a preprocessor performs the following activities:

--> Collects all the modules, files in case if the source program is divided into different modules stored at different files.

-->Expands short hands / macros into source language statements.

**Compiler:** Is a translator that takes as input a source program written in high level language and converts it into its equivalent target program in machine language. In addition to above the compiler also

--> Reports to its user the presence of errors in the source program.

--> Facilitates the user in rectifying the errors, and execute the code.

**Assembler:** Is a program that takes as input an assembly language program and converts it into its equivalent machine language code.

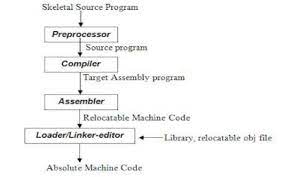
**Loader / Linker:** This is a program that takes as input a relocatable code and collects the library functions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

--> **Loading** consists of taking the relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.

--> **Linking** allows us to make a single program from several files of relocatable machine code. These files may have been result of several different compilations, one or more may be library routines provided by the system available to any program that needs them.

The steps involved in a typical language processing system can be understood with following diagram.

****

**3. Explain the phases of Compiler.**

## A. PHASES OF A COMPILER:

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. Following diagram (Figure1.4) depicts the phases of a compiler through which it goes during the compilation. There fore a typical Compiler is having the following Phases:

* 1. Lexical Analyzer (Scanner),
  2. Syntax Analyzer (Parser),
  3. Semantic Analyzer,
  4. Intermediate Code Generator(ICG),
  5. Code Optimizer(CO) , and
  6. Code Generator(CG).

In addition to these, it also has **Symbol table management**, and **Error handler** phases.

The Phases of compiler divided in to two parts, first three phases we are called as Analysis part remaining three called as Synthesis part.

The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.

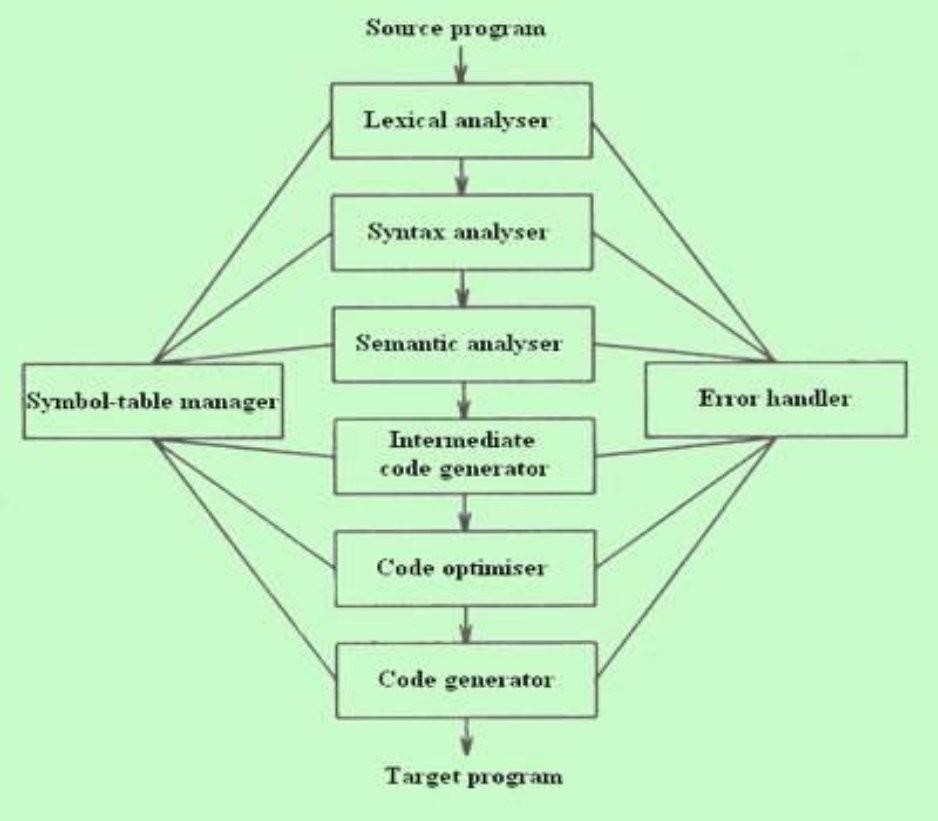
The analysis part is often called the front end of the compiler;

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.

In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly .



**Figure1.4 : Phases of a Compiler**

## PHASE, PASSES OF A COMPILER:

In some application we can have a compiler that is organized into what is called passes. Where a pass is a collection of phases that convert the input from one representation to a completely deferent representation. Each pass makes a complete scan of the input and produces its output to be processed by the subsequent pass. For example a two pass Assembler.

**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 belongs 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 DAG 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.

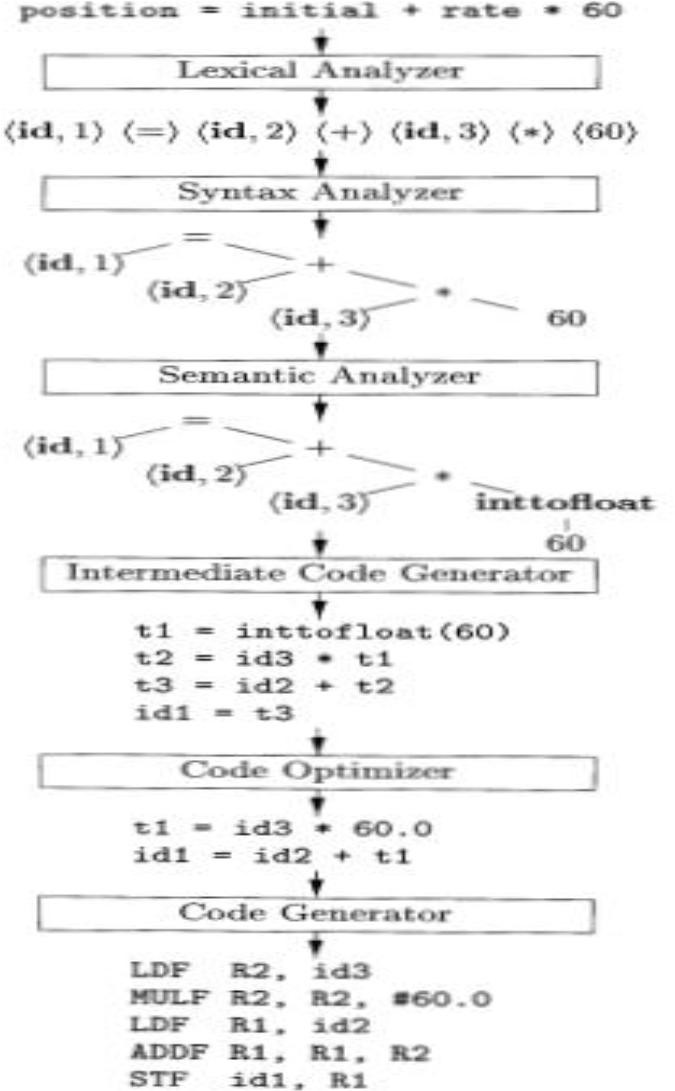
->Memory locations are selected for each variable used, and assignment of variables to registers is done.

-->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.

For example the source program is an assignment statement; the following figure shows how the phases of compiler will process the program.

The input source program is **Position=initial+rate\*60**



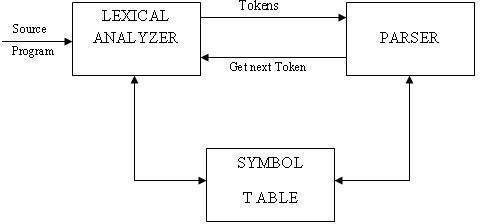
**4. What is the Lexical analyzer? Discuss the Functions of Lexical Analyzer.**

A. **OVER VIEW OF LEXICAL ANALYSIS**

* + - To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
    - Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

**ROLE OF LEXICAL ANALYZER**

the LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.



Upon receiving a get next token command form the parser, the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code, if the token is a simple construct such as parenthesis, comma or colon.

LA may also perform certain secondary tasks as the user interface. One such task is stripping out from the source program the commands and white spaces in the form of blank, tab and new line characters. Another is correlating error message from the compiler with the source program.

LEXICAL ANALYSIS VS PARSING:

|  |  |
| --- | --- |
| **Lexical analysis** | **Parsing** |
| A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc.  The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of  your grammar | A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence).  A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual  analysis). |

**5. Write short notes on tokens, pattern and lexemes?**

**A. TOKEN, LEXEME, PATTERN:**

**Token:** Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants

**Pattern:** 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.

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

Example:

Description of token

|  |  |  |
| --- | --- | --- |
| **Token** | **lexeme** | **Pattern** |
| const | const | Const |
| if | If | If |
| relation | <,<=,= ,< >,>=,> | < or <= or = or < > or >= or letter  followed by letters & digit |
| i | Pi | any numeric constant |
| nun | 3.14 | any character b/w “and “except" |
| literal | "core" | Pattern |

A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

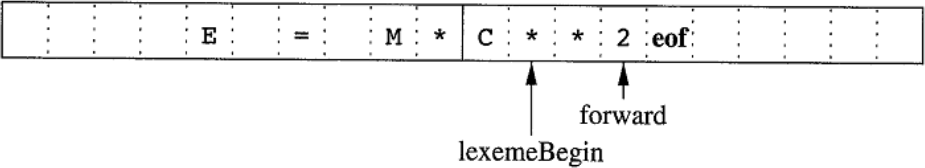
**6. Write short notes on Input buffering scheme? How do you change the basic input buffering algorithm to achieve better performance?**

## A. 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. 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 a two-character operator like ->, ==, or <=. Thus, we shall introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

### 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.



### Figure1.8 : 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 in to 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. The Pointer **lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.

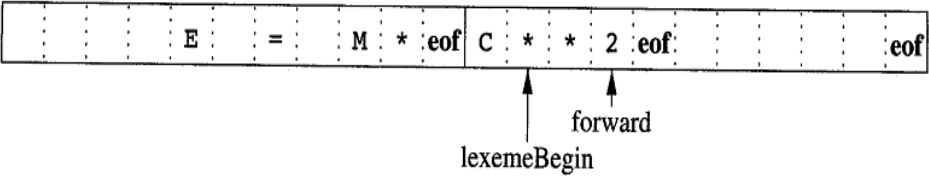
2. Pointer **forward** scans ahead until a pattern match is found;

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, 1exemeBegin is set to the character immediately after the lexeme just found. In Fig, 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.

### Sentinels To Improve Scanners Performance:

If we use the above scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multi way branch). We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a **sentinel** character at the end. The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**. Figure 1.8 shows the same arrangement as Figure 1.7, but with the sentinels added. Note that eof retains its use as a marker for the end of the entire input.



### Figure1.8 : Sentential at the end of each buffer

Any eof that appears other than at the end of a buffer means that the input is at an end.

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 )

{

break;}

**7. What do you mean by a Lexical analyzer generator? Explain LEX tool.**

A. Lex is a tool used to generate lexical analyzer, the input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler. Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called lex .yy .c, it is a c program given for C Compiler, gives the Object code. Here we need to know how to write the Lex language. The structure of the Lex program is given below.

**Structure of LEX Program :** A Lex program has the following form:

**Declarations**

**%%**

**Translation rules**

**%%**

**Auxiliary functions definitions**

**The declarations section** : includes declarations of variables, manifest constants (identifiers declared to stand for a constant, e.g., the name of a token), and regular definitions. It appears between %{. . .%}

In the **Translation rules** section, We place Pattern Action pairs where each pair have the form Pattern {Action}

**The auxiliary function** definitions section includes the definitions of functions used to install identifiers and numbers in the Symbol tale.

### LEX Program Example:

%{

/\* definitions of manifest constants LT,LE,EQ,NE,GT,GE, IF,THEN, ELSE,ID, NUMBER, RELOP \*/

%}

/\* regular definitions \*/

|  |  |  |
| --- | --- | --- |
| delim  ws | { | [ \t\n]  delim}+ |
| letter |  | [A-Za-z] |
| digit |  | [o-91 |
| Id |  | {letter} ({letter} | {digit}) \* |
| number |  | {digit}+ (\ . {digit}+)? (E [+-I]?{digit}+)? |
| %% |  |  |
| {ws} |  | {/\* no action and no return \*/} |
| If |  | {return(1F) ; } |

then {return(THEN) ; }

else {return(ELSE) ; }

(id) {yylval = (int) installID(); return(1D);}

(number) {yylval = (int) installNum() ; return(NUMBER) ; }

‖ < ‖ {yylval = LT; return(REL0P) ; )}

― <=‖ {yylval = LE; return(REL0P) ; }

―=‖ {yylval = EQ ; return(REL0P) ; }

―<>‖ {yylval = NE; return(REL0P);}

―<‖ {yylval = GT; return(REL0P);)}

―<=‖ {yylval = GE; return(REL0P);}

%%

**int** installID0() {/\* function to install the lexeme, whose first character is pointed to by yytext, and whose length is yyleng, into the symbol table and return a pointer thereto \*/

**int** installNum() {/\* similar to installID, but puts numerical constants into a separate table \*/}

**Figure : Lex Program for tokens common tokens**

### 8. List the compiler construction toolkit.

A. Compiler Construction Tools:

The compiler writer, like any software developer, can protably use modern software development environments containing tools such as language editors, debuggers, version managers, prolers, test harnesses, and so on.

In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler. These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms.

The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler.

Some commonly used compiler-construction tools include

1. Parser generators that automatically produce syntax analyzers from a grammatical description of a programming language.

2. Scanner generators that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. Syntax-directed translation engines that produce collections of routines for walking a parse tree and generating intermediate code.

4. Code-generator generators that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. Data- flow analysis engines that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data- flow analysis is a key part of code optimization.

6. Compiler-construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

### 9. Explain about Lexical errors.

.A. **LEXICAL ERRORS**:

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognise a *lexeme* as a valid *token* for you lexer. Syntax errors, on the other side, will be thrown by your scanner when a given set of **already** recognised valid tokens don't match any of the right sides of your grammar rules. simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error-recovery actions are:

1. Delete one character from the remaining input.
2. Insert a missing character in to the remaining input.
3. Replace a character by another character.
4. Transpose two adjacent characters.

### 10. Specify the functionality of linker, loader.

A. **Loader / Linker:** This is a program that takes as input a relocatable code and collects the library functions, relocatable object files, and produces its equivalent absolute machine code.

Specifically,

--> **Loading** consists of taking the relocatable machine code, altering the relocatable addresses, and placing the altered instructions and data in memory at the proper locations.

--> **Linking** allows us to make a single program from several files of relocatable machine code. These files may have been result of several different compilations, one or more may be library routines provided by the system available to any program that needs them.