Syntax Analysis and Semantic Analysis Part -1

Syntax and Semantic Analysis Part-1

Content :

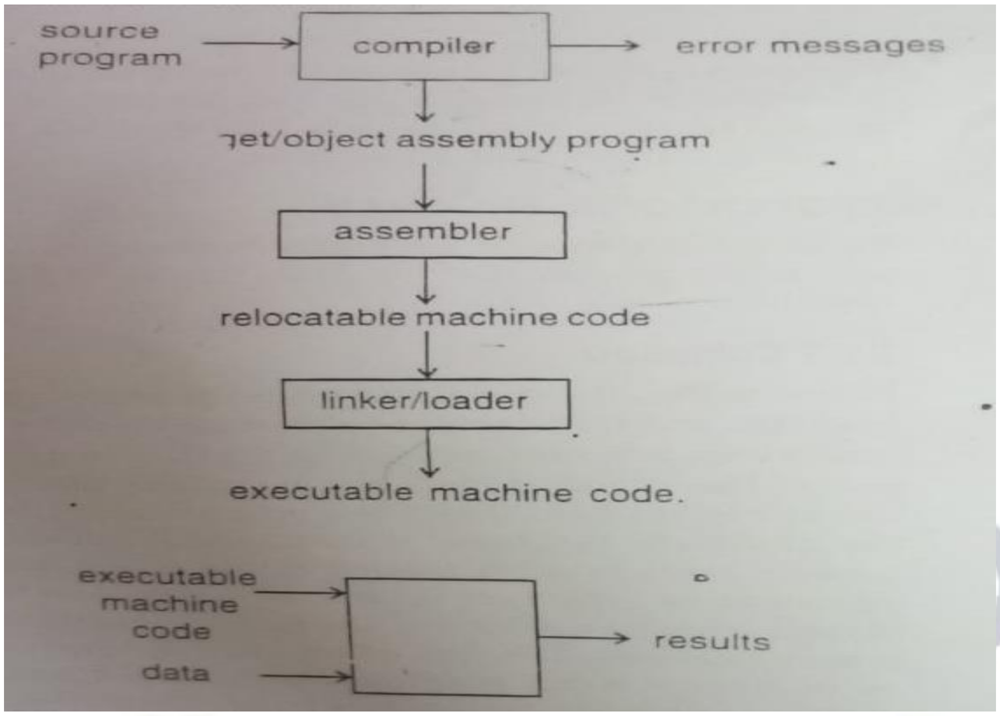
Debugger:

It is a computer program used by programmers to test and debug a target program. Debuggers may use instruction-set simulators, rather than running a program directly on the processor to achieve a higher level of control over its execution. This allows debuggers to stop or halt the program according to specific conditions. However, use of simulators decreases execution speed.

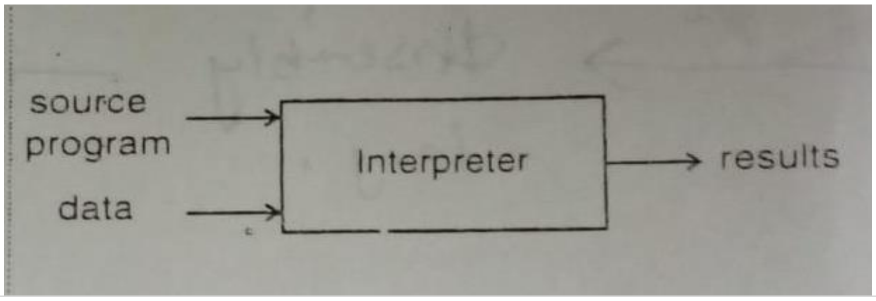
Compiler

The translator reads the source program written in one language and translates it to generate an object module which is an equivalent program written in another language----the target language. This object module is a re-locatable machine language form of the source program. There is a program called loader which performs the important functions of loading and link-editing. The process of loading consists of relocation of the object modules which means allocation of load time addresses which exist in the memory and placement of the relocated instructions and data in memory at the proper locations. The link editor links the object modules and allows us to make a single program from several files of re-locatable object modules to resolve mutual references. These files may be library files provided by the system which may be references. These files may be library files provided by the system which may be referenced by any program that needs them or they may be result of a different compilation.

Compiler: It translates a program written in high level language to generate assembly code or relocatable machine code and assembler translates an assembly program to generate relocatable machine code.



Interpreter: The data and source program are input to the interpreter. Producing any object module as in the compilation model, the interpreter produces the results by performing the operations of the source program on its data.



It is less efficient than execution of a compiled program as a source program has to be interpreted every time it is to be executed and this involves analysis of the source statement every time.

Parts of a compiler:

The fundamental language processing model for compilation consists of two step processing of a sour4ce program

- Analysis of a source program.
- Synthesis of a source program.

The analysis part breaks up the source program into its constituent pieces and determines the meaning of a source string and creates an intermediate representation of the source program. The synthesis part constructs an equivalent target program from the intermediate representation.

The three phases of analysis are:

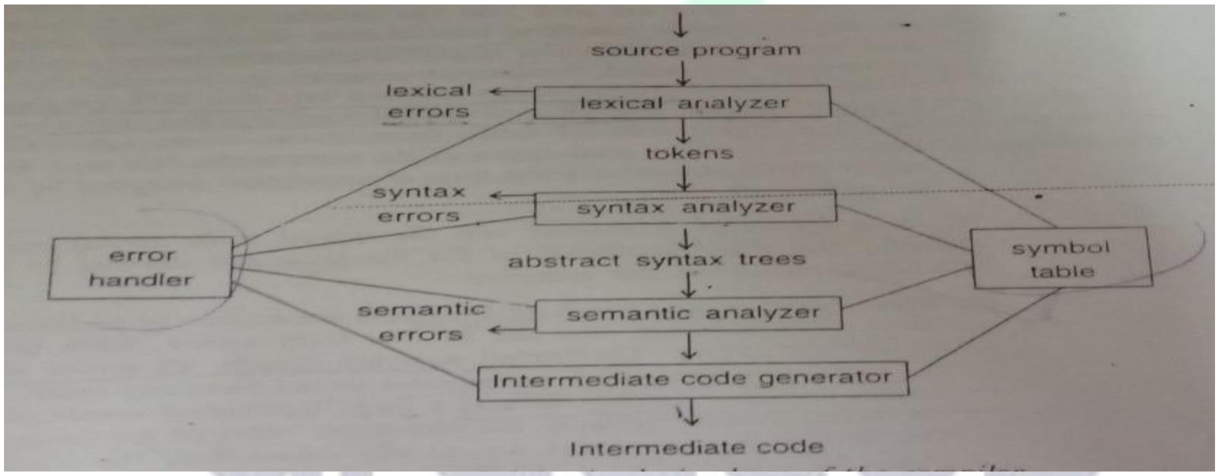
1. Lexical analysis which determines the lexical constituents in a source string by readingthe stream of characters from left to right and grouping them into tokens. A token is a sequence of characters having a collective meaning.
2. Syntax analysis which determines the structure of the source string by grouping the tokens into nested collections with collective meaning.
3. Semantic analysis which determines the meaning of the source string and ensures that the components of a program fit together meaningfully.

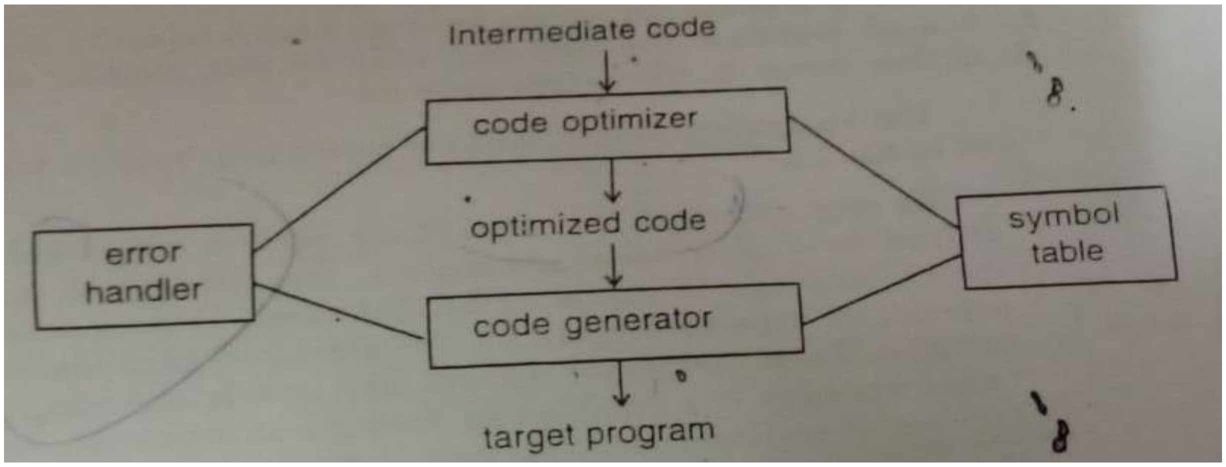
There are two other activities or phases which interact with all the other phases of the compiler. These are:

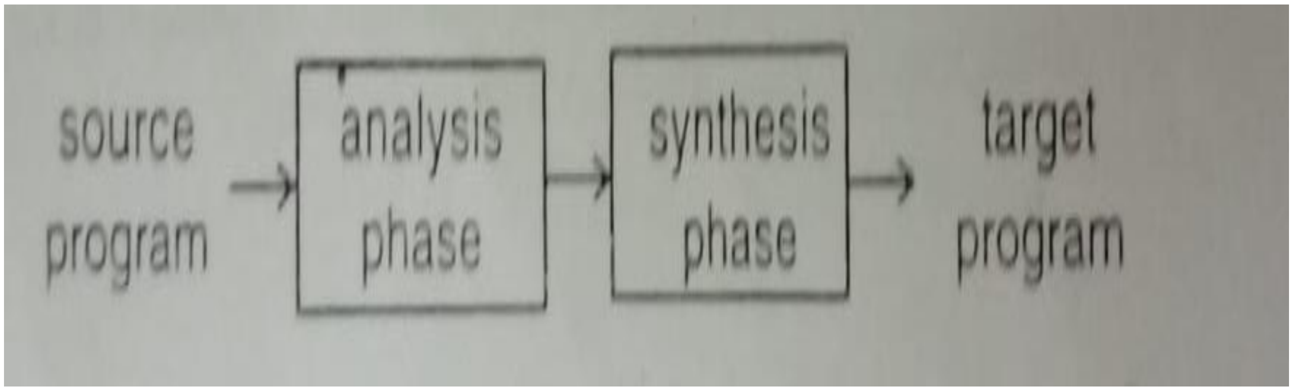
1. Symbol table management which records the identifiers used in the source program and also the information about various attributes of each identifier like its type, its scope the storage allocated for the identifier etc. Whenever the lexical analyzer detects an identifier in the source program, the identifier is entered into the symbol table which

is a data structure containing a record for each identifier where fields contain the attributes of the identifier.

2. Error detection and handling programs are written by programers and hence can not be free from errors. Each phase encounters errors. The lexical analyzer detects all errors when the remaining characters in the input do not form any token. The syntax analyzer usually detects a large number of errors where the token stream violates the structural rules of the language. The mismatch type errors are usually detected by the semantic analyzer.







Example of compilation

Consider the translation of the following statement

X=y ∗ z+10;

The internal representation of the source program changes with each phase of the compiler.

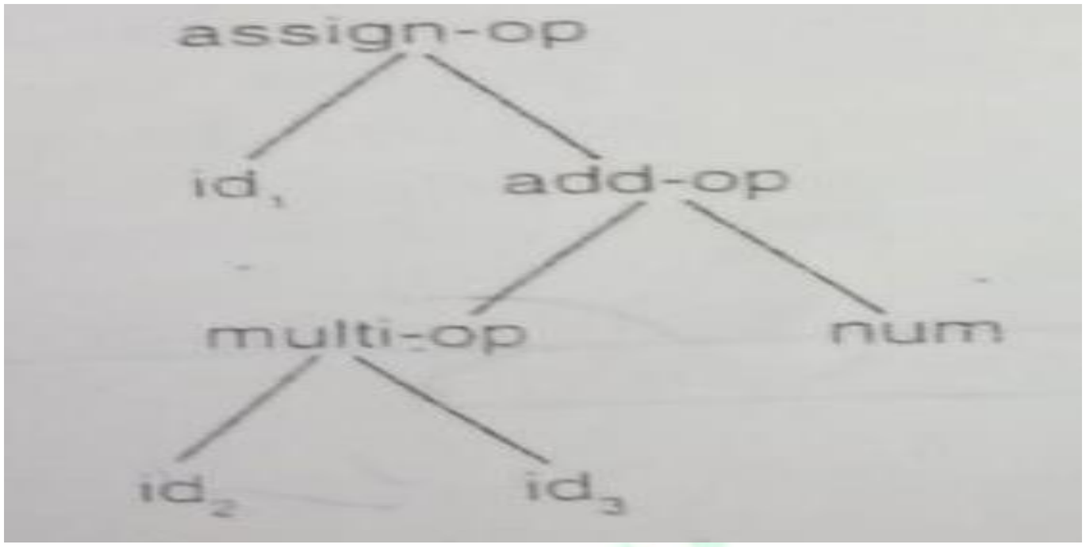
The lexical analyzer builds uniform descriptors for the elementary constituents of a source string.To do this, it must identify lexical units in the source string and categorize them into identifiers, constants or reserved words etc. The uniform descriptor is called a token. A token has the following format

category	Lexical value

A token is constructed for each identifier as well as for each operator in the string. Let it assign token id_1,id_2 and id_3 to x,y and z respectively, and assign -op to = , mult-op to ∗ , add-op to + and num to 10. After lexical analysis may be given as :

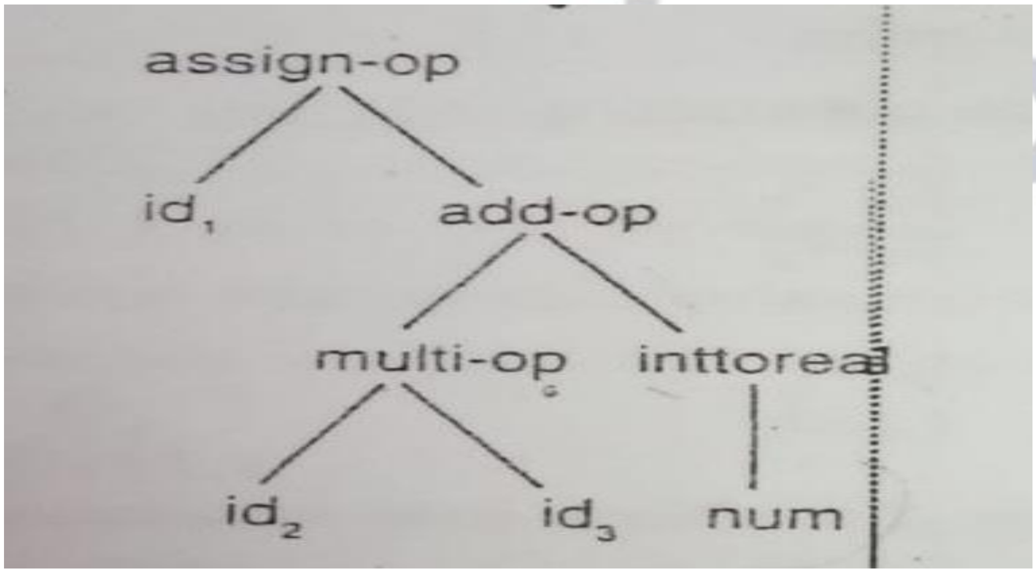
id_1 assign-op id_2 mult-op id_3 add-op num

The syntax analyzer determines the structure of the source string which may be represented by syntax trees.



Structure of x=y * z+10

The semantic analyzer determines the meaning of a source string. It finds out that the types of the operands are not identical as id_3 is real and num is integer. It converts num to type real so that the syntax tree now looks as:

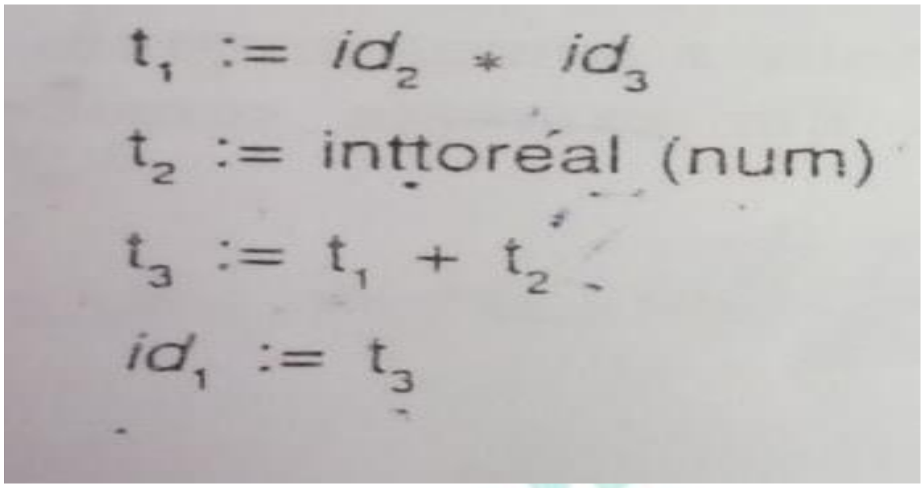


X=y *z+10 after semantic analysis

This intermediate code could be three address code or postfix code . But it should be easy to produce and easy to translate into the target program.

The following steps must be taken in the intermediate code

1. Multiply id_2 and id_3 in type real giving t_1.
2. Convert num to real giving t_2.
3. Add t_1 and t_2 in type real giving t_3.
4. Finally, store t_3 into id_1.



$t_1 := id_2 * id_3$

$t_2 :=$ inttoreal (num)

$t_3 := t_1 + t_2$

$id_1 := t_3$

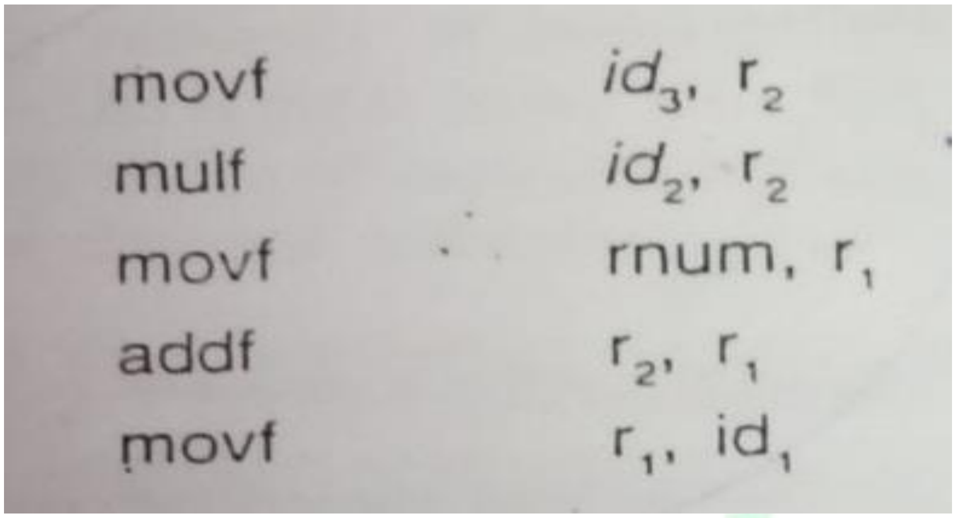
The code optimizer tries to improve the intermediate code in order to achieve faster running machine code. We can convert num from integer to real once and for all at compile time.

$T_1 := id_2 * id_3$

$Id_1 := t_1 +$ rnum(rnum is real equivalent of num)

Finally, the compiler can generate the target code for the intermediate code. The storage must be allocated or registers must be assigned to the variables and the addressing modes to be used for accessing the data must be decided before generating the code.

If we use two registers r_1 and r_2 the code can be written as :

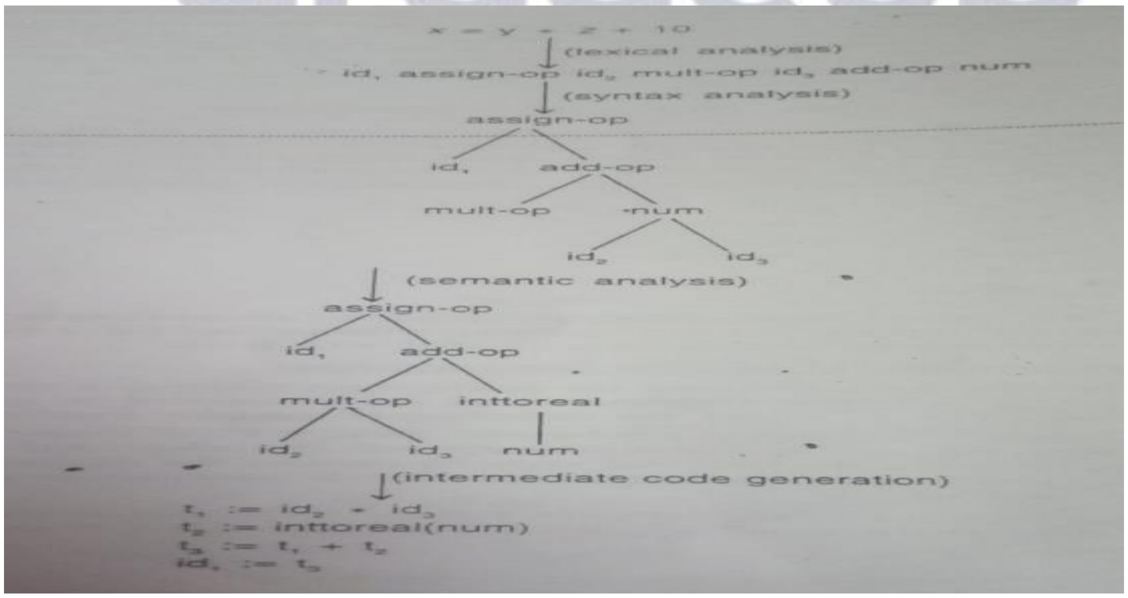


```
movf    id3, r2
mulf    id2, r2
movf    rnum, r1
addf    r2, r1
movf    r1, id1
```

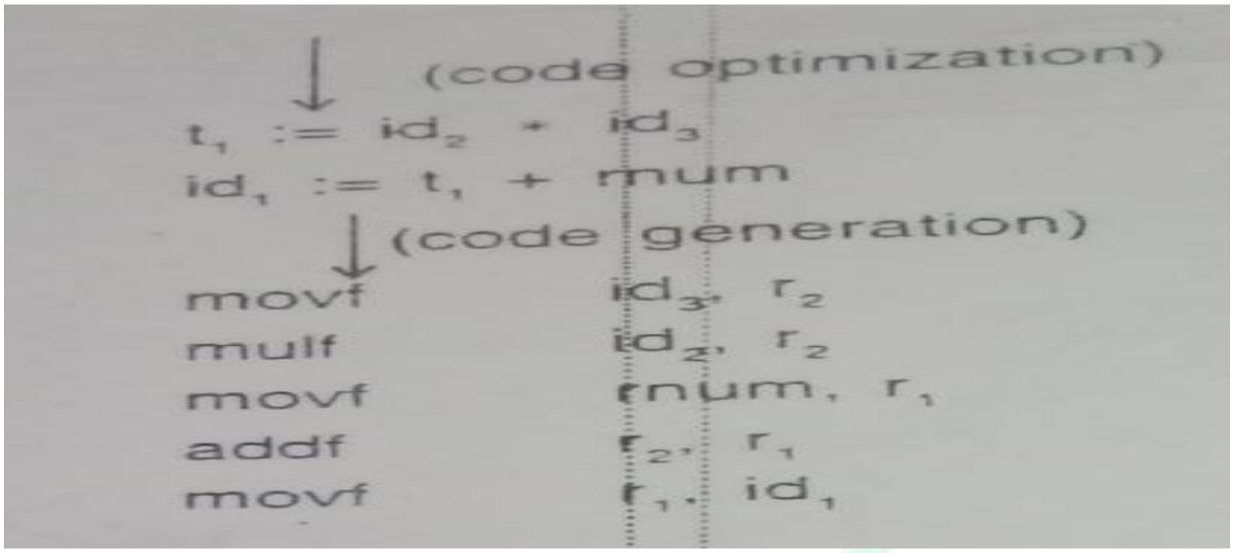
Contemporary Compiler Organization:

The front-end of the compiler includes those phases which are dependent on the source language and are independent of the target machine. It normally includes the analysis part. It may also include a certain amount of code optimization and the error handling.

The back-end of the compiler includes those phases which are dependent on the target language and are independent of the source language.



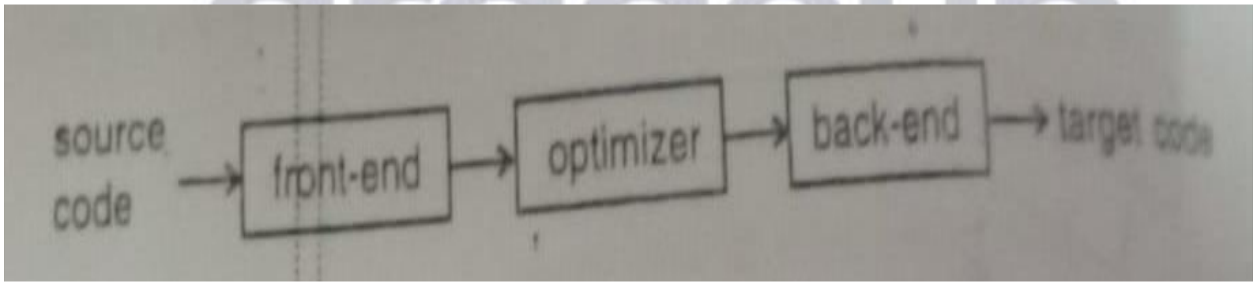
```
t1 := id2 * id3
t2 := inttoreal(num)
t3 := t1 + t2
id1 := t3
```



```
      ↓       (code optimization)
t1  := id2  *  id3
id1 := t1  +  num
       ↓ (code generation)
movf           id3, r2
mulf           id2, r2
movf           num, r1
addf           r2, r1
movf           r1, id1
```

The advantages of using this compiler organization are:

1. Take the front-end of a compiler and attach a different back-end to produce a compiler for the same source language on a different machine.
2. Take a common back-end for different front-ends to compile several different languages on the same machine.



$$\begin{tikzcd}
 & \text{Compiler} \arrow[dl] \arrow[dr] & \\
\text{Analysis} & & \text{Synthesis intermediate code}
\end{tikzcd}$$

Lexical analysis	Code optimization
Syntax analysis	Code generation
Symantec analysis	

↓

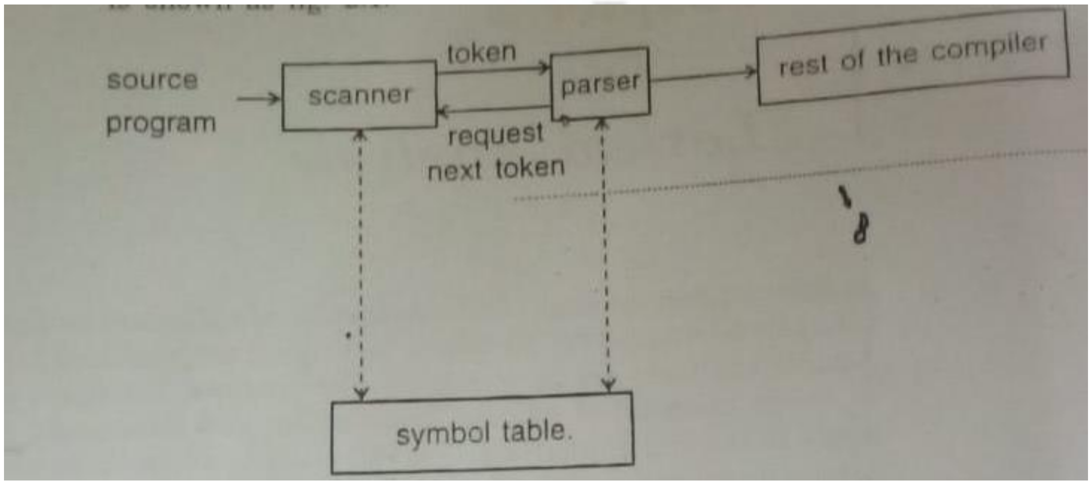
Role of Lexical Analyzer:

Lexical analyzer is the first phase of a compiler. Its job is reading the input program and breaking it up into a sequence of tokens. These tokens are used by the syntax analyzer. Each token is a sequence of characters that represents a unit of information in the source program.

It also performs certain secondary tasks like removing the comments and white spaces from the source program. It may also be given the responsibility of making a copy of the source program with the error messages marked in it. Each error message may also be associated with a line number. The analyzer may keep track of the line number by tracking the number of new line characters seen.

The relationship between lexical analyzer and the parser is well defined. The parser calls a single lexical analyzer subroutine every time it needs a new token and that subroutine reads input characters until it can identify the next token and returns it to the parser.

1. Generation of token
2. Removing the comments and white space.



3. Copy the source position with error.
4. Error must be linear.

Basic Definitions

Lexemes: They are the smallest logical units of a program. It is a sequence of characters in the source program for which a token is produced for example, if, 10.0, + etc.

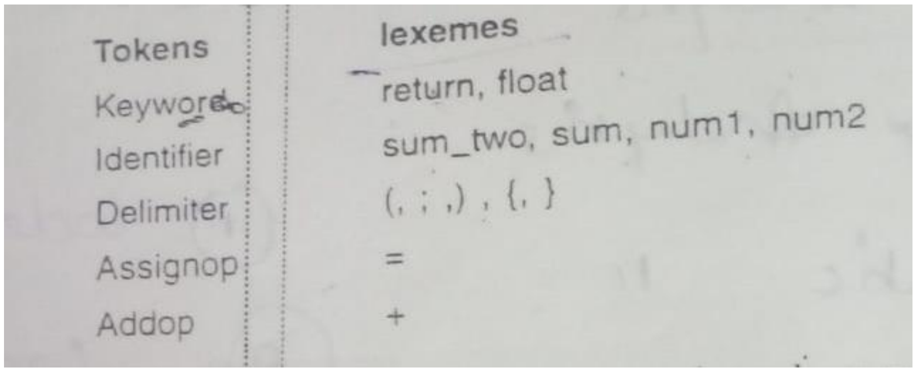
Tokens: classes of similar lexemes are identified by the same token. For example, identifier , number, reserve word etc.

Pattern: It is a rule which describes a token. For example an identifier is a string of at most 8 characters consisting of digits and letters. The first character should be a letter.

Example: Consider the following function definition in C.

```
Sum_two (float num1 , float num2)

{

Float sum;

Sum=num1 + num2;

Return(sum);
```

List of the tokens in the function definition and their corresponding lexemes.

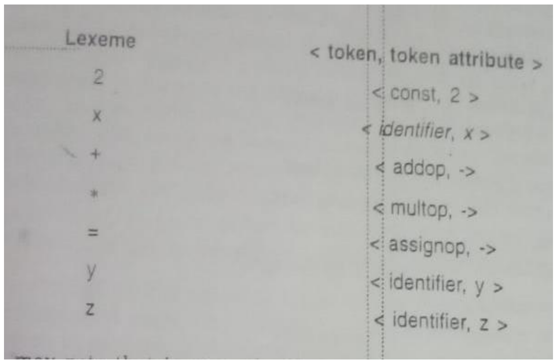


Tokens	lexemes
Keywords	return, float
Identifier	sum_two, sum, num1, num2
Delimiter	(, ; ,) , {, }
Assignop	=
Addop	+

User can treat each keyword, each operator and each delimiter as a separate token. The lexical analyzer must also pass additional indormation along with the token. These items of information are called attributes for tokens.

Example: List of token and their attributes for the following C statement:

X=y+z * 2



Lexeme	< token, token attribute >
2	< const, 2 >
x	< identifier, x >
+	< addop, ->
*	< multop, ->
=	< assignop, ->
y	< identifier, y >
z	< identifier, z >

Scanning The output

Lexical analyzer reads the source program character by character but reading the input character by character from the secondary storage is costly. Therefore, a block of data is first read into a buffer and then scanned by the lexical analyzer. One buffer scheme but it has some problems. If a lexeme crosses over the boundary of the buffer, the buffer has to be refilled in order to scan the rest of the lexeme and the first part of the lexeme will be overwritten in the process.

Another technique is two buffer scheme. In this scheme two buffers are scanned alternately. Each buffer is N character long, where N is the number of characters on one disk block. User read N input characters into each half of the buffer using one system read command. When one reaches the end of the current buffer, the other buffer is filled.

Token Specification

The methods for defining and recognizing patterns in strings of characters.

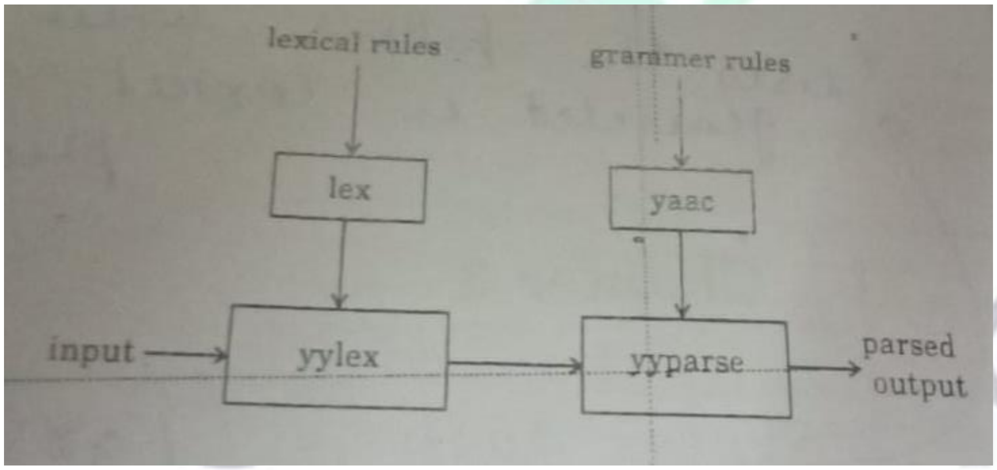
Regular expressions are important notation for specifying patterns. They represent patterns of strings of characters. It is completely defined by a set of strings which it matches. This set of strings is also called a language generated by the regular expression and is denoted as L(r).

An alphabet or a character class is a finite set of symbols. For example the set[0,1] is a binary alphabet or set of letters is a alphabet. An alphabet is usually written as a Greek symbol (sigma).

Automatic Generation of Lexical Analyzer:

User can generate lexical analyzer automatically by using a lexical analyzer generator. This generator takes as input the precise specification of the tokens of the language and a specification of the action to be performed on identification of each token. As an output, it generates a lexical analyzer. The unix tool lex is such a generator.

Lex: It is a lexical analyzer generator and yacc is a parser generator. They help us to write programs that transform structured input.



The lex specification: It is a series of statements of the form pattern action.

Lex patterns: They are more or less standard unix regular expressions, after the style of grep , sed , perl etc

Alpha_numeric: This includes all letters, digits, and _ , plus several others. They are, in effect, taken to be single character regular expressions.

A single character regular expression consisting of any one of the characters in between the square brackets[]

A range of characters may be specified by using a hyphen, like this

[A-Za-a]

To include the hyphen put it first or last in the group , like this[-A-Za-a]

To include a], put it first in the group, like this [] abc-]

If the first characters is ^ , it means any character except those listed.

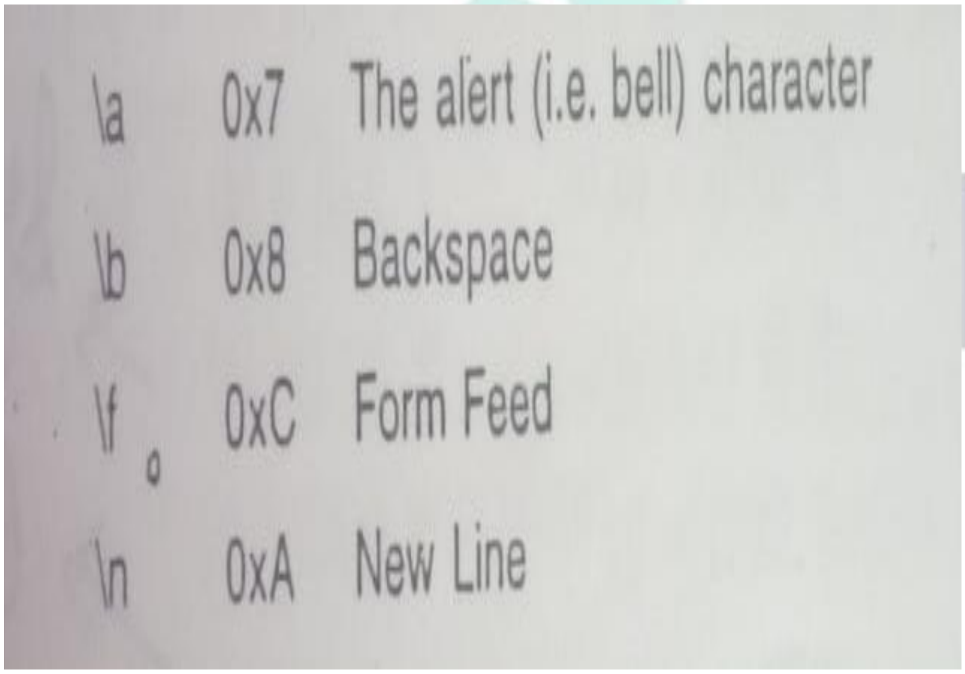
[^\n<>"]

Means "anything but a space, newline , quote, less than or greater than

\

Any character following the \ loses it's special meaning, and is taken literally. So the \ / \ / really means//

The \ is also used to specify the following special characters



\a	0x7	The alert (i.e. bell) character
\b	0x8	Backspace
\f	0xC	Form Feed
\n	0xA	New Line